Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Cybernetics

TEMPLATE-BASED ONTOLOGY EVOLUTION

Doctoral Thesis

Miroslav Blaško

Prague, September 2015

Ph.D. Programme: Electrical Engineering and Information Technology
Branch of study: Artificial Intelligence and Biocybernetics

Supervisor: Dr. Zdeněk Kouba

Abstract

Evolving ontologies by domain experts is difficult and typically cannot be performed without the assistance of an ontology engineer. This process takes a long time and often recurrent modeling errors have to be resolved. This doctoral thesis proposes a technique for creating controlled ontology evolution scenarios that ensure consistency of possible ontology evolutions and give guarantees to the domain expert that his/her updates do not cause inconsistency. We introduce ontology templates that formalize the notion of controlled evolution together with a consistency checking algorithm. Ontology templates are reusable across multiple scenarios which helps ontology engineers to manage whole ontology evolution. In addition, the algorithm for consistency checking can be used within an auxiliary reasoning service that provides ontology engineers with new options to check certain expectations of the evolving ontology. The specific contributions of the thesis are: (1) framework and methodology for definition of ontology evolution scenarios, (2) algorithms to compute ontological implications of the evolution scenarios, (3) proposal of new non-standard reasoning service that implements the algorithms to support ontology engineering process, (4) prototypical implementation of the service and its experimental evaluation, and (5) validation of the approach on real cases.

Keywords:
   ontology template, ontology evolution, reasoning, consistency checking, ontology engineering, semantic web.

Acknowledgements

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

With the growing amount of data, formal ontologies are becoming more and more popular for capturing the shared meaning of heterogeneous, incomplete data as well as subsequent data integration. These features distinguish open-world ontologies from closed-world relational databases. Ontologies provide more expressive constructs to capture data meaning, not only at instance level (particular cases) but also at schema level (general knowledge). Logic-based formalization of ontologies is exploited during automated reasoning, in particular for inferring new knowledge, powerful consistency checking [1], query answering [2], or error explanation [3].

Ontology engineering (and ontology design in particular) is a complex engineering discipline, requiring a deep knowledge of the particular domain as well as a familiarity with top-level ontologies, existing domain ontologies and design methodologies. During ontology design, domain experts and computer scientists cooperate. As a result of some ontology engineering methodologies like Methontology [4], a *core schema* is designed in cooperation with both parties, possibly with the use of *ontology design patterns* [5]. However, an ontology is a dynamic body and evolves in time as new knowledge becomes available. The evolution process is tackled in existing ontology engineering methodologies at a high level, requiring – among others – ontology consistency to be maintained.

In order to make ontology evolution efficient, domain experts can be given tools to enrich the ontology themselves. For example, in the MONDIS project [6, 7] we came across the scenario where domain experts had to develop taxonomical knowledge. Although they were able to perform this task (e.g. they developed the taxonomy of building components and materials), they were not able to judge the computational impact of the evolution. Due to the unknown nature of the evolution, ontology consistency has to be checked on the fly which slows the work down. Even worse, in the case where ontology enrichment results in an inconsistency/unsatisfiability or any other undesired inference, it has to be debugged and explained by standard techniques [3]. This process is typically hard for domain experts to understand and interpret and requires the assistance of computer scientists during all ontology evolution scenarios.

To cope with these problems, we introduce the idea of *controlled ontology enrichment* and formalize it into the notion of *ontology templates* – a compact representation of controlled evolution scenarios. Ontology templates define simple ontology evolution scenarios for which the domain expert can be sure that the knowledge he/she creates does not cause inconsistency. A possible impact of the ontology templates on the ontology consistency is precomputed in advance, before the domain expert starts evolving the ontology.

We will explain the motivation as well as formalization of our framework with the real-case example of the MONDIS project. Within the project, a core ontology [8] for the description

of monuments was created and later enriched with MONDIS-specific taxonomies. The domain experts were asked to add general knowledge about the developed taxonomies using specific relations of the core ontology. An example of such a relation is the "hasMaterial" relation, expressing that a building component (object described by the concept "Component") is made of a building material (object described by the concept "Material"). Using the relation it can be stated at a terminological level which building component type may/may not be associated with which building material type. For example, it can be said that an object of type pillar may be made of wood, but may not be made of glass. Within the thesis, we will illustrate, using this running example how this specific evolution scenario can be formalized by ontology templates *and how it can be used to check its consistency in advance – thus possibly providing human-readable explanations or fixes of the modeling errors.*

1.1 Thesis contributions

This thesis proposes a novel technique in the field of ontology engineering. The technique suggests breaking down parts of an ontology evolution into small evolution scenarios based on interaction with domain experts. Each ontology evolution scenario is defined in terms of atomic additions and restrictions on their use. Atomic additions are organized by higher order statements which we call ontology templates. Ontology templates are also used to formulate restrictions on use of those atomic operations. With such formalization we provide a semi-automatic approach for creation of correct and complete set of restrictions applicable to the scenario. As a consequence we are able to guide domain expert through the evolution explaining him precomputed consequences of his actions. Ontology templates play important role not only in explanation of restrictions to domain experts, but also provide a way to share and reuse higher order statements accross the scenarios. We argue and show on real cases that it is useful to organize part of the knowledge using ontology templates. It can help to domain experts as well as ontology engineers to check their expectations about consequences of an ontology evolution. This technique can be implemented to existing ontology editors, or scripting tools as will be demonstrated. Moreover, the output of this formalization can be directly used to guide model-driven Semantic Web applications. The specific contributions of this thesis are :

- ○ framework and methodology for definition of ontology evolution scenarios

- ○ algorithms to compute ontological implications of the evolution scenarios

- ○ proposal of new non-standard reasoning service that implements the algorithms to support ontological engineering methods

- ○ prototypical implementation of the service and its experimental evaluation

- ○ validation of the approach on real cases

1.2 Structure of the text

Chapter 2 reviews works related to our approach and introduces the necessary background on formal ontology languages, error explanation and privacy protection languages in order to understand the rest of the text. Chapter 3 presents the formal model of ontology evolution, ontology template validation and reasoning services and methodoloty to use ontology templates. Chapter 4 presents the prototypical implementation of the ideas and evaluation of the implementation. Chapter 5 demonstrates the techniques on two use cases. The thesis is concluded by Chapter 6.

2 Background and State-of-the-Art

This chapter is divided into two sections. The first section describes the theoretical background needed to understand the following chapters. The second section describes related work to our approach.

2.1 Theoretical Background

This section provides an overview of the ontological background needed to understand the following sections as well as technologies that are used in our framework for privacy protection. P3P is used to specify privacy policies of services while SemPref specifies privacy preferences of users. The creation of privacy policies and privacy preferences is guided by semantic technologies OWL and SPARQL-DL. An ontology, within the scope of this thesis, is a specification of conceptualization [9] – an abstract and simplified view of the world shared between applications or people. It uses declarative logic-based formalism to describe entities and relationships between those entities within the domain of interest. Statements about the domain are called axioms.

2.1.1 $\mathcal{ALC}$ DL - Syntax and Semantics

Current ontology standards, like OWL 2 [10], are based on description logics. Description logics are monotonic and decidable subsets of first-order predicate logic typically aimed at knowledge representation. Although our framework is not dependent on any particular variant of Description Logics ($\mathcal{DL}$), we will use the $\mathcal{ALC}$ language [11], basic description logic capturing the fundamentals of more expressive formalisms used in the semantic web domain.

An ontology is represented in terms of *individuals* (representing *particulars*, i.e. concrete objects), *concepts* (representing *universals*, i.e. set of objects) and *roles* (representing relations between objects).

$\mathcal{ALC}$ **Syntax**   The vocabulary of $\mathcal{ALC}$ consists of three disjoint sets of *concept names* (N_C), *role names* (N_R), and *individual names* (N_I). Let A be a concept name and R a role name. Any $\mathcal{ALC}$ concept can be constructed inductively by the following syntax rules:

$$
\begin{aligned}
C \rightarrow &A \mid \top \mid \bot \mid \neg C \mid \exists R.C \mid \forall R.C \mid \\
&C_1 \sqcap C_2 \mid C_1 \sqcup C_2
\end{aligned}
\tag{2.1}
$$

An ontology consists of a finite set of axioms – *concept inclusion axioms* of the form $C_1 \sqsubseteq C_2$, *class assertions* of the form $C(a)$, and *role assertion* of the form $R(a, b)$.

$\mathcal{ALC}$ **Semantics**   The formal semantics of $\mathcal{ALC}$ is defined by *interpretation* $\mathcal{I}$ which is a pair $\langle \Delta^{\mathcal{I}}, \cdot^{\mathcal{I}} \rangle$. A non-empty set $\Delta^{\mathcal{I}}$ denotes the domain of interpretation. $\cdot^{\mathcal{I}}$ is an interpretation function that maps each concept name $A \in N_C$ to a subset of $\Delta^{\mathcal{I}}$, and each role name $R \in N_R$ to a subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$. The interpretation function is extended for concepts in the following way :

$$\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$$
$$\bot^{\mathcal{I}} = \emptyset$$
$$(\neg C)^{\mathcal{I}} = \Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$$
$$(C_1 \sqcap C_2)^{\mathcal{I}} = (C_1)^{\mathcal{I}} \cap (C_2)^{\mathcal{I}}$$
$$(C_1 \sqcup C_2)^{\mathcal{I}} = (C_1)^{\mathcal{I}} \cup (C_2)^{\mathcal{I}}$$
$$(\exists R.\top)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} \mid \exists b.\langle a, b \rangle \in R^{\mathcal{I}}\}$$
$$(\forall R.C)^{\mathcal{I}} = \{a \in \Delta^{\mathcal{I}} \mid \forall b.\langle a, b \rangle \in R^{\mathcal{I}} \rightarrow b \in C^{\mathcal{I}}\}$$

We say that an ontology O is *consistent* if there is an interpretation $\mathcal{I}$ such that for every axioms α, $\mathcal{I}$ entails α. Formally, $\forall \alpha \in O, \mathcal{I} \vDash \alpha$ such that:

$$\mathcal{I} \vDash (C_1 \sqsubseteq C_2) \in O \quad \text{iff} \quad (C_1)^{\mathcal{I}} \subseteq (C_2)^{\mathcal{I}}$$
$$\mathcal{I} \vDash C(a) \in O \quad \text{iff} \quad a^{\mathcal{I}} \in C^{\mathcal{I}}$$
$$\mathcal{I} \vDash R(b, c) \in O \quad \text{iff} \quad \langle b^{\mathcal{I}}, c^{\mathcal{I}} \rangle \in S^{\mathcal{I}}$$

An interpretation $\mathcal{I}$ that satisfies O is called the *model* of O. An ontology O entails an axiom α if every model of O satisfies the axiom α. O entails an ontology O' if every model of O is also model of O'.

In addition to ontology consistency and entailment checking, more complicated reasoning services that make use of consistency checking are necessary in practical cases. We will need the error explanation service that can explain ontology inconsistency in terms of a *minimal inconsistency preserving set* (MIPS)[1].

Formally, for an inconsistent ontology O, a MIPS is a set of axioms $\mu \sqsubseteq O$ which is *inconsistent*, and *minimal* with this property, i.e. $\forall \alpha \in \mu : (\mu \setminus \{\alpha\})$ is consistent. Generally, an inconsistent ontology can have many "roots of inconsistency" corresponding to many MIPS-es. Computing all of them is an exponential problem in the number of ontology consistency checks [13]. Fortunately, computing a single MIPS is polynomial, as shown in [13]. We will use this technique for computing a single MIPS during evolution schema consistency checking in Section 3.3.

In addition, we will need a technique for solving a well known NP-Complete problem, i.e. finding a *minimal hitting set* [14] [15] of some $\mathcal{X}$, where $\mathcal{X}$ is a set of subsets of a finite set S. A *minimal hitting set* of $\mathcal{X}$ is a minimal set S' such that $S' \subseteq S$ and S' contain at least one element

---

[1]In the literature, concept unsatisfiability, or general entailment explanations are often considered [12] instead of an ontology consistency explanation. We will use just the latter notion, as both concept unsatisfiability and entailments can be reduced to ontology inconsistency problems as shown in [1].

from each subset in $\mathcal{X}$. In Diagnosis subfield of artificial intelligence, this problem refers to finding finding minimal diagnoses of conflict set. Moreover, in this thesis we will be computing minimal diagnosis based on conflict sets represented by MIPS of an ontology.

2.1.2 P3P

P3P is an XML-based language that enables services to describe their privacy practices in a structured computer-readable form. It comes in two versions: P3P 1.0 [16], which became a W3C recommendation in April 16, 2002 and P3P 1.1 [17] which has been a working group note since November 16, 2006. Each P3P policy file may contain one or more policies. Each policy is specified by a POLICY element and may include the following sub-elements:

- ENTITY - identifies the legal entity making a representation of privacy practices

- ACCESS - indicates whether the site allows users to access the various kinds of information collected about them

- DISPUTES-GROUP - describes dispute resolution procedures to be followed when disputes arise about service's privacy practices

- EXTENSION - describes an extension to the syntax

- STATEMENT - describes what data practices are applied to what data types

The POLICY element may contain multiple STATEMENT elements. Each statement defines what data (e.g. user's name) and data categories (e.g. user's health data) are being collected as well as for which purposes (e.g. historical preservation), recipients (e.g. public fora) and retention (e.g. for the stated purpose) that define the attributes of the collection. Thus, the STATEMENT element may contain the following sub-elements:

- DATA-GROUP - specifies what data will be collected from a user

- PURPOSE - describes for what purposes the collected data are used

- RECIPIENT - describes with whom the collected data will be shared

- RETENTION - describes how long the collected data will be held for

For each of the above sub-elements of the STATEMENT element a predefined set of values and extension mechanism to define new values is specified. For example, some of the predefined values (subelements of PURPOSE) of the PURPOSE element are:

- current - completion and support of activity for which the data were provided

- individual-analysis - doing research and analysis that uses information about the user

- telemarketing - contacting visitors for the marketing of services or products via telephone

  ∘ contact - contacting visitors for the marketing of services or products

Each of the purpose values can also have an attribute required that accepts the following values:

  ∘ opt-in - the user has to affirmatively request usage of data for this purpose

  ∘ opt-out - the data may be collected for this purpose unless the user requests otherwise

  ∘ always - the data will be used for this purpose (this is the default value).

The DATA-GROUP element contains one or more DATA sub-elements. A DATA element refers to one or more datatypes (data identifiers). An example of such a datatype that describes the user's family name would be `#user.name.family`. Most of the predefined data types are grouped into one or more data categories, which provide another way to refer to collected data. For example, data type `#user.name.family` or `#user.home-info.postal.street` (user's street address) are both associated with the category physical (user's physical contact information). The "optional" attribute of DATA can be used to state that a particular data element collection is optional, otherwise (and by default) it is required.

An example of a P3P policy is shown in Figure 2.1. The first statement in the policy expresses that the datatype `#user.home-info` (user's home contact information) is collected for the purposes contact and individual-analysis. The usage of collected data for contact purpose may be opt-out by user while for the individual-analysis purpose it is required. According to the second statement, the only data that cannot be opt-out for contact purpose is `#user.home-info.online.email` (user's email address) and `#user.name` (user's name). The recipient of all the collected data is ours (i.e. the data is shared with the service's entity).

The major problem of P3P is its lack of formal semantics. The solution to this issue proposed by [18] defines two possible semantics compatible with P3P specification: data-centric and purpose-centric relational semantics. Both semantics are based on the translation of P3P policies into relational schema. There is an expressiveness vs. simplicity trade off between the data-centric and the purpose-centric semantics of P3P policies. Data-centric semantics is more coarse-grained, however it has sufficient expressiveness for the use in our privacy scenario, thus from this point on we will stick to it.

Data-centric semantics uses a relational schema with five relations (d-purpose, d-recipient, d-retention, d-collection, d-category). The relational schema of the relations taken from [19] is shown in Figure 2.1.2. Given a set of P3P statements it is straightforward to translate them to the corresponding database content: for each data type item in a P3P statement, one needs to pair it with purpose, recipient and retention and assign it to the corresponding relations. Note that [19] does not address all P3P elements such as access, entity and disputes-group, but it is very simple to extend it to handle these attributes as well, as their semantics is rather weak in P3P. For the sake of simplicity, we will not consider those attributes in the rest of this paper.

Although a P3P policy file has to comply with the P3P XML schema file defined by W3C, it can contain semantic inconsistencies. Many of the inconsistencies were identified in [20, 21]. Additionally, Li et al. [18] characterized similar types of inconsistencies and created three classes of integrity constraints that should be fulfilled by every P3P policy:

```
<POLICY>
  <STATEMENT>
    <DATA-GROUP>
      <DATA ref="#user.home-info"/>
    </DATA-GROUP>
    <PURPOSE>
      <contact required="opt-out"/>
      <individual-analysis required="always"/>
    </PURPOSE>
    <RECIPIENT>
      <ours required="always"/>
    </RECIPIENT>
  </STATEMENT>
  <STATEMENT>
    <DATA-GROUP>
      <DATA ref="#user.home-info.online.email"/>
      <DATA ref="#user.name" />
    </DATA-GROUP>
    <PURPOSE>
      <contact required="always"/>
    </PURPOSE>
    <RECIPIENT>
      <ours required="always"/>
    </RECIPIENT>
  </STATEMENT>
</POLICY>
```

Figure 2.1: an example of a P3P policy. The first statement expresses that the datatype #user.home-info is collected for purpose contact whose usage may be opt-out and purpose individual-analysis, whose usage is always required. The second statement says that the datatype #user.home-info.online.email (user's email address) and #user.name (user's name) is always required for the contact purpose. The recipient of all the collected data is ours (i.e. the data is shared with the service's entity).

○ Data-centric constraints: the keys in the relations defining semantics generates four functional dependency constraints. For example, in the d-purpose relation, a pair (data, purpose) uniquely defines a value of the required field of the relation (see Figure 2.1.2). Consider the policy example in Figure 2.1. The data-centric constraint would be violated if we added a new statement that collects the #user.home-info datatype for the contact purpose that is opt-in (contact purpose would become both opt-in and opt-out for the same data - #user.home-info).

Relation name	Field name	Allowed values	Key for the relation
d-purpose	data purpose required	URI reference to datatype The P3P-defined purpose values opt-in, opt-out, always	(data, purpose)
d-recipient	data recipient required	URI reference to datatype The P3P-defined recipient values opt-in, opt-out, always	(data, recipient)
d-retention	data retention	URI reference to datatype The P3P-defined retention values	(data)
d-collection	data optional	URI reference to datatype required, optional	(data)
d-category	data category	URI reference to datatype The P3P-defined category values	(data,category)

Table 2.1: The schema of basic relations according to data-centric semantics

○ Data-hierarchy constraints: datatypes in P3P are organized in hierarchies (we will speak about children, resp. parents and descendants, resp. ascendants for this P3P data hierarchy traversal description). Each node represents a particular datatype. The P3P semantics specifies that whenever a policy statement refers to a datatype represented by the node it may also collect any datatype of its descendant nodes. Consider again the first statement of the policy from Figure 2.1. The `#user.home-info` datatype is collected for the contact purpose that may be opt-out by the user. It is reasonable to assume that a service may collect some more specific datatype such as `#user.home-info.postal` or `#user.home-info.online` for the contact purpose that is "at least opt-out" (i.e. the required attribute should be opt-out or always, but not opt-in). Thus, a service should be allowed to add a new statement to the policy in Figure 2.1, that e.g. collects the `#user.home-info.online.email` datatype for contact purpose which is always required (or opt-out) (see the second statement in Figure 2.1). Based on the above observation, Li et al. [18] define a total ordering "<" for the values of the required attribute of PURPOSE values such that opt-in < opt-out < always. Using the ">" relation the above constraint can be formulated as follows: for d-purpose(d1, p1, r1) and d-purpose(d2, p2, r2), if d1 is more specific than d2 and p1 = p2, then r1 ≥ r2 must hold. A similar ordering was defined for values of the relation d-recipient, d-collection and d-retention.

○ Semantic vocabulary constraints: some P3P predefined values are related to other P3P pre-defined values in a way that one value may forbid or request the usage of the other predefined value. As an example of a statement violating such a constraint, consider a statement that collects data for purpose "historical" with retention value "no-retention". The purpose "historical" implies that the collected information is going to be archived, while "no-retention" implies that collected information must not be logged, archived or otherwise stored.

In the following text we will describe how semantic technologies can be used to create a framework that is able to identify any inconsistencies in P3P policies. In order to define the inconsistencies more clearly we will define three classes of restrictions that the P3P framework should consider according to the scope in which they should be satisfied:

○ statement-scope restrictions: include all restrictions that must be satisfied in a P3P statement without knowing the context of the statement (i.e. the whole P3P policy). As an example of such a restriction, consider the first statement of the example policy depicted in Figure 2.1. In the statement a service states that the collection of `#user.home-info` data is required (the default value of the "optional" attribute is "no"). Thus it is reasonable to assume that at least one purpose and one recipient should be declared as always required. Otherwise, it would be unclear weather `#user.home-info` data may be collected (but not used) or it should be not collected at all. Thus, if we change the statement's purpose individual-analysis or recipient ours to be opt-in or opt-out, the statement would become inconsistent according to this statement-scope restriction.

○ global-scope restrictions: include all restrictions that must be satisfied in the P3P model and do not depend on a particular P3P policy. An example of such a restriction, presented in section Section 5.2.1, shows the category propagation semantics.

○ policy-scope restrictions: include all restrictions that are neither statement-scope restrictions nor global-scope restrictions. An example of such a restriction is the following semantic vocabulary constraint: the stated-purpose retention in a P3P policy signals that the referenced data will be retained for the period necessary to meet the stated purposes. However, if multiple purposes of the datatype are specified it may lead to confusion. In such cases, it seems reasonable to have just one purpose value. Another example of policy-scope restrictions are data-centric constraints explained above.

2.1.3 Sempref

Privacy preferences in SemPref are expressed, similarly to APPEL, using two kinds of rules: accept rules and reject rules. Each rule has a head and a body. A rule head defines the behavior of the rule i.e. accept or reject. The rule body is a set of constraints that specifies conditions on data, purpose and recipient, etc. Each constraint consists of zero or more predicates that either test a set membership (i.e. one can use it to express either that an element is in a set or that an element is not in a set) or check a value of an attribute. An example of a complex reject rule introduced in [18] is in Figure 2.2. The rule in the figure rejects policies that provide personal information (datatypes from categories physical, demographic and uniqueid) to third parties, but only when the collection of such information is required or data sharing is always.

The reject rule in the figure contains two statements. Each of the statements contains three predicates. The relationship among multiple predicates in constraints is logical and.

2.1.4 OWL and SPARQL-DL

Although the SemPref language uses traditional relational database technology to provide well-defined semantics to (a rather vague specification of) P3P privacy policies, it still turns out that one can easily produce a privacy policy that is semantically inconsistent. Due to their flexibility and high expressiveness, semantic web technologies, and especially the Web Ontology Language

reject:

[data-category ∈ {physical, demographic, uniqueid}
  collection=required
  recipient ∉ {ours}]
[data-category ∈ {physical, demographic, uniqueid}
  recipient ∉ {ours}
  recipient-required ∈ {always}]

Figure 2.2: An example of two SemPref rules that reject policies which provide personal information (datatypes `physical`, `demographic`, `uniqueid`) to third parties, but only when the collection of such information is required (first rule) or data sharing is `always` (second rule). Predicates in the rule body are connected as logical *and*.

OWL [22], seem to be an appropriate formalism to capture these semantic inconsistencies. OWL became a W3C recommendation in 2004 and since then, many knowledge modeling application areas adopted it as a first-choice knowledge representation language. From the knowledge modeling point of view, its main difference to classical relational database technology is the open world assumption (OWA) that - together with full logical negation - makes it a perfect fit for a distributed and incomplete web environment. OWA, however, disallows using rule engine for sound and complete reasoning. Additionally, OWL is richer in expressiveness providing a bunch of additional constructs, like subsumption relations between unary predicates (classes) and binary predicates (properties), transitivity and symmetry of properties, inverse properties or cardinality restrictions. The upcoming standard OWL 2 [23] raises the expressiveness even more to support e.g. property composition, qualified cardinality restrictions and complex data ranges.

Unfortunately, there is no standard language/engine to query OWL ontologies that would be widely accepted by the community. The first-choice option for query answering in OWL ontologies is often the W3C recommendation for querying Resource Description Framework (RDF), the SPARQL language. However, SPARQL is primarily targeted at querying RDF which is significantly weaker in expressiveness than OWL. This might cause incomplete results w.r.t OWL semantics when used to query OWL ontologies. A solution that is adopted in this paper is to use SPARQL syntax to pose SPARQL-DL queries. SPARQL-DL [24, 25] is a language extending standard conjunctive data queries (analogous to SQL) by constructs expressing meta-queries. These meta-queries are especially useful when building complex ontologies with an expressive and complex Tbox part, which is our case. Using SPARQL-DL we can easily traverse the policy ontology described below.

2.1.5 SPARQL Inferencing Notation (SPIN)

Effective management of SPARQL (SPARQL-DL) queries is important when there are multiple queries that share common parts. Although, the SPARQL standard [26] does not address this issue, there is the language SPIN (SPARQL Inferencing Notation) [27] that can be used for this

purpose. SPIN is a collection of RDF vocabularies (currently a W3C Member Submission) enabling the use of SPARQL to define constraints and inference rules on Semantic Web models. It also provides meta-modeling capabilities that allow users to define their own SPARQL functions and query templates. TopBraid Composer [28] is an editor for Semantic Web ontologies that provides comprehensive support for SPIN, including an inference engine, editors and parsers.

The SPIN API [2] is an open source Java API that provides (1) converters between textual SPARQL syntax and the SPIN RDF Vocabulary, (2) a SPIN-based constraint checking and inferencing engine, (3) support to execute user-defined SPIN functions and templates. The SPIN API is built on the Apache Jena [29] which is one of the most mature frameworks to build Semantic Web applications.

As an example of SPIN serialization consider query that evaluate true if there is a person older than 18 years. Such a query can be represented by SPARQL ASK query :

```
# must be at least 18 years old
ASK WHERE {
     ?person my:age ?age .
     FILTER (?age < 18) .
}
```

The serialization of this query in SPIN is as follows:

```
[ a sp:Ask ;
        rdfs:comment "must be at least 18 years old"^^xsd:string ;
        sp:where ([ sp:object sp:_age ;
                        sp:predicate my:age ;
                        sp:subject spin:_person
                   ] [ a       sp:Filter ;
                        sp:expression
                                [ sp:arg1 sp:_age ;
                                  sp:arg2 18 ;
                                  a sp:lt
                                ]
                   ])
   ]
```

2.1.6 Activities related to ontologies

Activities related to ontologies that are relevant to our work can be defined according to [30] as follows:

- *Ontology management* is a set of methods and techniques that support creating, modifying, versioning, querying, and storing ontologies.

○ *Ontology modification* refers to the activity of changing the ontology without considering its consistency of ontologies.

○ *Ontology evolution* refers to the activity of facilitating the modification of an ontology by preserving its consistency.

2.2 Related Work

This section provides an overview of related work in the area of template-based management and second-order reasoning over ontologies.

Ontology Pre-Processor Language (OPPL) [31] is a domain-specific macro language for the manipulation of ontologies written in OWL. There exists a Java API to use the language within an application as well as plugins for the major OWL editor Protege [32]. The plugins allow for defining macros – operations (such as add/remove axiom, rename entity) on an ontology, which are parameterized by variables. Variable values can be assigned to the macros by OPPL query, or manually by setting the values in a form. The OPPL Patterns plugin [33] provides a mechanism to find out whether the macro-based modification of an ontology is safe – it cannot change the interpretation of existing symbols within the ontology. The analysis outputs which variables can take symbols from the signature of the ontology and which variables must be outside the signature in order for the modification to be safe. This provides a certain insight into the possible effect of the macro for an ontology engineer before applying it.

Another approach [34] using OPPL defines common anti-patterns that can be used as additional information to ontology debugging services. When a modeling error occurs, anti-patterns can help in explaining the error at a higher level. If there are more explanations of modeling errors, the explanations are prioritized such that anti-pattern based explanations are used first.

A major RDF-based ontology editor TopBraid Composer integrates SPIN [27] templates to parameterize operations over ontologies. SPIN templates are parameterized SPARQL queries that can be serialized into RDF. They are useful to attach additional validation constraints or inference rules to OWL classes, in order to validate/infer over their instances. However, validation constraints at the axiom level are by design not very convenient to use.

The approach [35] uses Description Logics with Temporal Logic operators to formally characterize and reason about ontology evolution. The approach allows for expressing some statements about previous snapshots of an ontology (i.e. whether a concept was satisfiable in any snapshot or all previous snapshots).

There are many approaches that extend Description Logic [36, 37, 38] in order to provide reasoning with higher-level constructs. However, reasoning is incomplete for second-order logic, thus they provide either too weak extension [39] towards second-order or they extend too simple description logic (such as [40]). Hence such approaches are not applicable to our use case.

The paper [41] provides a second-order framework and related calculus to unify and solve many non-standard reasoning tasks in Description Logics such as concept unification, concept contraction etc.

Computationally, our approach closely relates to minimal hitting set problem or to one of its relatives such as the minimum set cover problem, and especially in model-based diagnosis and Reiter's [14] first principles. It describes technical systems as composed of several components that may cease to operate as designed. The discrepancy between the expected and observed behavior of the system results from the malfunctioning of one or more components. The minimal diagnoses refers to finding smallest set of faulty components that needs to be replaced for the system to work. Reiter showed that diagnoses, i.e. minimal hitting sets of conflict sets can be computed by HS-trees. Greiner [42] has revised HS-tree into an HS-DAG to provide efective pruning of the HS-tree. BHS-tree algorithm based on a binary-tree and Boolean algebra algorithm was intruduced by [43] and then later optimized by Pill et al. [44]. Another interesting aproach based on binary matrix is provided by Staccato [45]. We will use Reiter's, Staccato, BHS-tree and both variants of Boolean algorithms within the thesis as optional variant to compute hitting sets. An empirical evaluation of the algorithms is published in [46].

Our approach in most cases will have large number of small conflict sets, which might be applicable for kernelization techniques (such as [47]) of so called d-hitting set problems, where d represents restriction on size of conflict set. Such techiques are however not applicable in cases where input is minimal conflict set, which will be our case.

Current methodologies for ontology engineering, like Methontology [4], Neon methodology [48], On-to-knowledge methodology [49], and Uschold and KingâĂŹs methodology [50] try to guide ontology engineer in various phases of the ontology lifecycle. Yet, they are not specific w.r.t. the ontology engineering activity itself – it is completely in the hands of the ontology engineer. Comparing to them, our approach focuses on systematization and optimization of the recurring process of ontology enrichment. In this sense, my approach is complementary to the existing ontology engineering methodologies.

3 Formalization of Ontology Evolution

The goal of this chapter is the formalization of ontology evolution, motivated by Section **??**, in which it would be possible to (1) describe controlled ontology evolution by atomic operations, (2) predict ontological implications of such operations in advance, and (3) manage multiple ontology evolution scenarios. The first section of this chapter defines a conceptual model for ontology evolution scenarios. The second section defines a formal model for the computation of the ontological implications. The third section introduces algorithms to compute such ontological implications, proves their correctness and completeness, and describes their computational complexity. Most of the content of the second and the third section was published also in [51]. The last section of this chapter provides methodology for the use of the provided conceptual model in order to manage evolution scenarios effectively, taking into account computational limits of the approach.

3.1   Conceptual Model for Controlled Ontology Evolution

An ontology evolution starts from an initial ontology that is modified by the addition and removal of axioms. In a *controlled ontology evolution* scenario we assume that each atomic operation of the evolution adds/removes a predefined set of axioms called *ontology template grounding*, where *ontology template* defines a family of such sets. Moreover, we assume that the initial ontology will be the stable part of the evolution, i.e. no axioms from this ontology will be removed during the evolution. We will call this stable initial ontology *core ontology* of the evolution.

Recall the description of ontology activities from Section 2.1.6. In the same sense with respect to the scenario from the previous paragraph we will use terms (1) *template-based ontology modification* – an ontology modification using *ontology templates*, when the consistency of the ontology is not considered, and (2) *template-based ontology evolution* – an ontology modification using *ontology templates*, when the consistency of the ontology is considered. In addition, we will use the term *template-based ontology extension* to refer to both an activity and an ontology that was created from a *core ontology* by the addition of some *ontology template groundings*.

The goal of this conceptual model for a controlled ontology evolution is to guide the user within the process. Although many atomic operations (in terms of additions and removals) could be applied during the process, it is not important to track all those operations within the model. The evolved ontology on which we need to build a guidance is a set of axioms and the order in which the axioms were added is irrelevant for our purpose. Without a loss of generality we could even assume that the evolution is a sequence of atomic additions only. Removals will not need
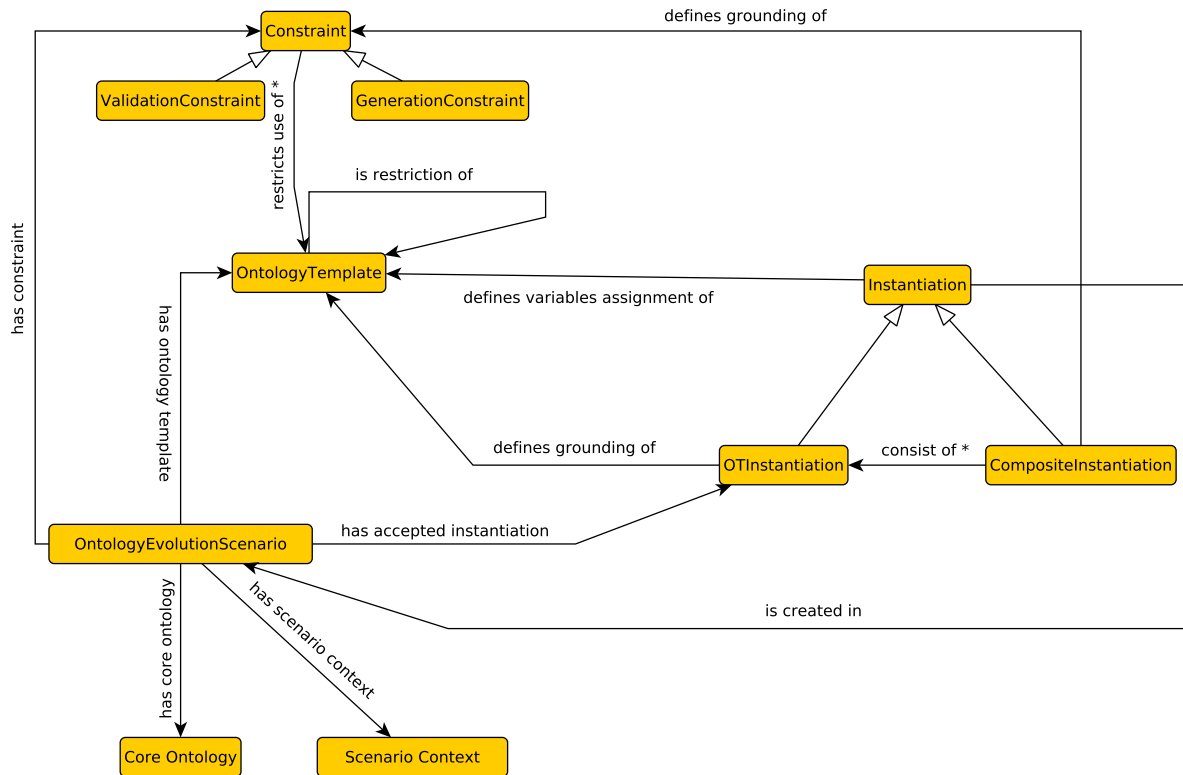
Figure 3.1: Conceptual model for controlled ontology evolution

any guidance as they will not cause any ontological issues due to the open world assumption. On the other hand it is useful to track which atomic additions form the current state of the evolved ontology. We will refer to them as *instantiations of ontology templates*. Each ontology template instantiation represents some template grounding (set of axioms) of one ontology template. Note that one set of axioms can be potentially generated from different template groundings thus instantiations provide a way to track their origin.

Finally, to provide guidance on which ontology template instantiations can be added based on the state of the evolution we define the terms *constraint* and *constraint instantiation*. Constraint instantiation defines which sets of *ontology template instantiations* cannot be used together, while constraint defines some family of those sets. There are two reasons why constrains are applied to controlled ontology evolution. First, some applications of ontology template instantiations could cause ontological consequences that are undesired. An example would be the inconsistence of the ontology or the unsatisfiability of its concepts. Second, the constraint is outside the scope of the ontology that is being evolved. This happens for example when the ontology language is not expressive enough to encode this constraint, or this constraint does not apply to every use of this ontology.

Another issue that our conceptual model needs to solve is the manageability of multiple ontology evolution scenarios. *Ontology templates* are higher level constructs that are meant to be

reused across multiple scenarios. Based on the concrete evolution scenario each *ontology template* is used to generate a set of *ontology template instantiations*. In order for ontology templates to be reusable to some extent we should separate the context from which instantiations are generated from the definition of a template. In most cases, the context from which the instantiations are generated is *core ontology*. Our proposal is however to optionally include part of "generation procedure" in the definition of an ontology template. Thus for each ontology evolution scenario we will define a *scenario context* which will be used in combination with ontology templates to generate its instantiations. Next, we will define some relations between concrete ontology templates that allow the reuse of ontology templates by restricting a set of groundings they generate.

Based on the above requirements we created the conceptual model presented in Figure 3.1. As an example of its usage, consider the example about building components and materials. The core ontology could be Monument Damage Ontology, the scenario context could be some external ontology that defines building components and materials. An ontology template could be represented by query that creates the groundings from the external ontology.

In the following section, we restrict our scope to define ontology templates and related terms to provide reasoning services.

3.2 Formal Model for Reasoning

As stated in the introductory part of this thesis, there are reasons for the inclusion of additional semantics to parts of the ontology and in order to support extended reasoning, it must be done in a formal way.

First, the grammar of logic-based ontology languages is typically designed to find a compromise between contradicting requirements such as support for reasoning, simplicity of grammar constructors, and ease of adoption by software tools. For example, to support reasoning, ontology languages restrict the grammar to constructs that in most or all cases make reasoning decidable. Thus, the basic building block of an ontology language, an ontology axiom, is usually not convenient to express some statement in a person's mind. Especially in cases where the ontology contains a lot of complex and recurring structural patterns, it is easier to think of such patterns as wholes.

Second, there are many cases where the possible evolution of ontology by an application can be described in terms of atomic operations on the ontology. If we are able to formalize the possible space of the evolution, we can apply reasoning techniques of the source language to provide inference or debugging services of the evolving ontology in advance.

The formal model able to describe the mentioned evolution and its reasoning services must be able to describe: the initial setting from which ontology evolution starts; atomic operations of the evolution using some higher level constructs ("a template"), and a way to express new inference and debugging services in terms of defined atomic operations.

An ontology evolution starts from an initial ontology that is being modified by the addition and removal of axioms. We assume that each atomic operation of the evolution adds/removes a predefined axiom set called *axiom template grounding*, where *axiom template* defines a family of such sets. Moreover, we assume that the initial ontology will be the stable part of the evolution,

i.e. no axioms from this ontology will be removed during the evolution. This stable initial ontology we call the *core ontology* of the evolution.

In the following text we define described terms formally.

3.2.1 $\mathcal{ALC}_x$ Syntax

Ontology Template We denote $N_X = \{X_1, X_2, ..\}$ a set of concept name variables[1]. We define language $\mathcal{ALC}_x$ by extending $\mathcal{ALC}$ $\mathcal{DL}$ as follows. A *concept template* is an expression defined by rules for $\mathcal{ALC}$ concepts (2.1) extended with the new rule $C \to X$, where $X \in N_X$. An *axiom template* is defined the same way as the $\mathcal{ALC}$ axiom in Section 2.1.1 with *concept templates* in place of *concepts*. Similarly, an *ontology template* is a finite set of *axiom templates*. We call the axiom/ontology template *ground* if it does not contain any variable. A function $\Upsilon : N_X \to \mathcal{P}(N_A)$ is a *variable domain function* – it maps each concept name variable to a subset of all concept names.

Ontology Template Set An *ontology template set* is a set $\Lambda = \{T_i \mid 1 \le i \le n_\Lambda\}$ such that:

○ each $T_i = \{t_{ij} \mid 1 \le j \le n_{T_i}\}$ is an *ontology template* that consists of *axiom templates t_{ij}*

○ $var(t_{ij})$ is a set of all variables occurring in the signature of axiom template t_{ij}

○ $var(T_i) = \bigcup var(t_{ij})$ is a set of all variables occurring in ontology template T_i

○ $var(\Lambda) = \bigcup var(T_i)$ is a set of all variables occurring in Λ

Recall the motivating example from the introductory chapter and consider the ontology of building components and materials that contains four axioms (we will use the abbreviation *OM* for concept *OrganicMaterial* and *VE* for concept *VerticalElement* later in the text):

$$O_{CM} = \{Pillar \sqsubseteq VerticalElement,$$
$$VerticalElement \sqsubseteq Component,$$
$$Wood \sqsubseteq OrganicMaterial,$$
$$OrganicMaterial \sqsubseteq Material\}$$

$\Lambda_{CM} = \{T_1, T_2\}$ is an ontology template set such that:

○ $T_1 = \{\$COMP_1 \sqsubseteq \forall hasMaterial.\neg\$MAT_1 \}$

○ $T_2 = \{\_c : \$COMP_2 \sqcap \exists hasMaterial.\$MAT_2 \}$

$\_c$ is an anonymous individual [52]. Ontology template T_1 states that all individuals of type $\$COMP_1$ must have material that does not belong to concept $\$MAT_1$. Ontology template T_2 states that there exists some individual of type $\$COMP_2$ that has material of type $\$MAT_2$. Υ is a variable domain function such that:

---

[1]Although I define the whole framework for concept name variables to cover the most typical ontology evolution scenario. Yet, the framework can be easily extended to support also individual variables or role variables.

- $\Upsilon(\$COMP_1) = \Upsilon(\$COMP_2) = \{Pillar, VerticalElement, Component\}$

- $\Upsilon(\$MAT_1) = \Upsilon(\$MAT_2) = \{Wood, OrganicMaterial, Material\}$

Substitution Set   Intuitively we define the notion of *substitution* σ and a set of *substitutions* $\Sigma = \{\sigma_1, \sigma_2, ..\}$. In the context of an *ontology template T* these notions will help us to differentiate between one grounding of *T* and all possible groundings of *T*. In the context of an *ontology template set* these notions help us to differentiate between one grounding of all templates from Λ and all possible ontology template groundings that define possible evolutions w.r.t. Λ. The concrete definitions are:

- $var \subseteq N_X$ – a set of variables

- $\sigma : var \rightarrow N_A$ is a *substitution* (*variable substitution function*) – a function from a subset of concept variables to concept names. We will use the notation: $\sigma = \{[X_1/A_1], ..., [X_n/A_n]\}$, where $[X_i/A_i]$ means that σ maps variable X_i to the concept name A_i; $dom(\sigma)$ for the domain of σ.

- σ is *ground substitution* w.r.t. *T* if $var(\sigma(T)) = \emptyset$. We call $\sigma(T)$ *grounding* of axiom template w.r.t. σ and define *template*(σ) = *T*. Similarly, *ground substitution* can be extended to Λ.

- σ_{wg} is the *weakest ground substitution* of σ w.r.t. *T* if $\forall \sigma'(\sigma'$ is ground substitution and an extension of $\sigma) \implies \sigma'(T) \models \sigma_{wg}(T)$

- Σ is a set of substitutions

- Σ is a *ground substitution set* w.r.t. *T* if every $\sigma \in \Sigma$ is ground w.r.t. *T*. Similarly the definition can be extended to Λ.

  As an example, consider substitution $\sigma` = \{[\$COMP_1/Pillar]\}$ and ground substitutions $\sigma_1, \sigma_2, \sigma_3, \sigma_4$ such that:

  - $\sigma_1 = \{[\$COMP_1/Pillar], [\$MAT_1/Wood]\}$
  - $\sigma_2 = \{[\$COMP_1/VE], [\$MAT_1/OM]\}$
  - $\sigma_3 = \{[\$COMP_2/Pillar], [\$MAT_2/OM]\}$
  - $\sigma_4 = \{[\$COMP_2/VE], [\$MAT_2/Wood]\}$

  $\sigma`$ is not ground substitution w.r.t. $T_1$ as $dom(\sigma`) = \{\$COMP_1\}$ does not contain $\$MAT_1$. σ_1, σ_2 are ground substitutions w.r.t. T_1, while σ_3, σ_4 are ground substitutions w.r.t. T_2. Thus *template*(σ_1) = *template*(σ_2) = T_1. $\{\sigma_1, \sigma_2\}$ and $\{\sigma_1, \sigma_3\}$ are examples of ground substitution sets w.r.t. T_1 and Λ, respectively. $\{\sigma`, \sigma_2\}$ is not a ground substitution set w.r.t. T_1 or T_2.

Instantiation Set   In order to define possible extensions of a *core ontology* and an *ontology template set*, we define the notion of an *instantiation set*. Intuitively, any possible extension of the *core ontology* w.r.t. *ontology template set*, i.e. an ontology O' can be defined by one instantiation. O' is the result of applying the instantiation to the *core ontology* and *ontology template set*. Let O be a *core ontology*, Λ be an *ontology template set*, Σ_i a substitution set for each i:

- A substitution Σ is an *instantiation* w.r.t. Λ, if every $\sigma \in \Sigma$ is a ground substitution of an *ontology template T* from Λ such that $dom(\sigma) = var(T)$.

- $\mathcal{S} = \{\Sigma_1, \Sigma_2, ..\Sigma_i\}$ is an *instantiation set* w.r.t. Λ if each non-empty Σ_i is an instantiation w.r.t. Λ.

- $cl(\mathcal{S})$ is a *closure of $\mathcal{S}$ under inclusion*, i.e. $cl(\mathcal{S}) = \{\Sigma' \mid \Sigma \in \mathcal{S} \land \Sigma' \subseteq \Sigma\}$. If $\mathcal{S} = cl(\mathcal{S})$ we say that instantiation $\mathcal{S}$ is *closed under inclusion*.

- $max_{\subseteq}(\mathcal{S})$ is a set of all maximal elements of $\mathcal{S}$ w.r.t. a partially ordered set ordered by inclusion.

- An *instantiation set* is *complete* if it is a power-set of some substitution set, i.e. $\mathcal{S} = \mathcal{P}(\Sigma)$.

Any subset of $\{\sigma_1, \sigma_2, \sigma_3, \sigma_4\}$ is an instantiation w.r.t. Λ, while the substitution set $\{\sigma_1, (\sigma_2 \cup \sigma_3)\}$ is an example of a ground substitution set which is not an instantiation w.r.t. Λ. $\mathcal{S}' = \{\{\sigma_1, \sigma_2\}, \{\sigma_1, \sigma_3\}, \{\sigma_1\}\}$ is an example of an instantiation set, while its closure under inclusion is the set $cl(\mathcal{S}') = \{\{\sigma_1, \sigma_2\}, \{\sigma_1, \sigma_3\}, \{\sigma_1\}, \{\sigma_2\}, \{\sigma_3\}, \emptyset\}$ and $max_{\subseteq}(\mathcal{S}') = \{\{\sigma_1, \sigma_2\}, \{\sigma_1, \sigma_3\}\}$

Instantiation function   Intuitively, an *instantiation function* applies substitutions to relevant ontology templates. For ground substitutions it returns a list of axioms, otherwise it returns list of partially substituted axiom templates. Let T be an *ontology template*, Λ be an *ontology template set*, an *instantiation function IF* is defined as follows :

- $IF(\sigma, T) = \sigma(T)$

- $IF(\Sigma, \Lambda) = \{\alpha \mid \alpha \in IF(\sigma, T) \land (\sigma \in \Sigma) \land (T \in \Lambda) \land (dom(\sigma) \cap var(T) \neq \emptyset)\}$

- $IF(\mathcal{S}, \Lambda) = \bigcup_{\Sigma \in \mathcal{S}} IF(\Sigma, \Lambda)$

For example, $IF(\sigma`, T_1) = \{Pillar \sqsubseteq \forall hasMaterial.\neg \$MAT_1\}$, while $IF(\sigma_1, T_1) = IF(\{\sigma_1\}, \Lambda) = IF(\{\{\sigma_1\}\}, \Lambda) = \{Pillar \sqsubseteq \forall hasMaterial.\neg Wood\}$

Modification graph   Let $\mathcal{S}$ be some instantiation set representing all possible extensions of a *core ontology O* w.r.t. *ontology template set* Λ. A modification graph of $\mathcal{S}$ is an oriented graph where:

- each node represents one instantiation from $\mathcal{S}$,

Figure 3.2: Complete modification graph w.r.t. a canonical evolution schema $\langle O, \Lambda, \mathcal{S}_{\mathcal{G}}, \mathcal{S}_{\mathcal{V}} \rangle$, green (red) sub-graph represents $\mathcal{S}_{\mathcal{G}}$ ($\mathcal{S}_{\mathcal{V}}$), respectively.

- ○ each edge connects an instantiation from a node representing Σ_i to a node representing Σ_j if there exists a non empty substitution σ such that $(\sigma \notin \Sigma_i) \wedge (\Sigma_i \cup \{\sigma\} = \Sigma_j)$.

We say that a modification graph is complete if it is a graph of a complete instantiation set. Figure 3.2 shows a modification graph for the complete instantiation set which is a power-set of four grounding substitutions – $\sigma_1, \sigma_2, \sigma_3, \sigma_4$. In the context of an ontology extension activity w.r.t. O and Λ, the bottom node with label "{}" refers to a core ontology O, while the top node with label "$\{\sigma_1, \sigma_2, \sigma_3, \sigma_4\}$" refers to an ontology that was created by the addition of four ontology template groundings – $O \cup IF(\{\sigma_1, \sigma_2, \sigma_3, \sigma_4\}, \Lambda)$. Each edge of a graph can be viewed as addition or removal (if opposite direction is considered) of one ontology template grounding.

Evolution schema For a template-based ontology extension activity we define an *evolution schema* as a 4-tuple $ES = \langle O, \Lambda, \mathcal{S}_{\mathcal{G}}, \mathcal{S}_{\mathcal{V}} \rangle$ such that

- ○ O is a *core ontology*

- ○ Λ is an *ontology template set*

- ○ $\mathcal{S}_{\mathcal{G}}$ is a *generating instantiation set* – an instantiation set that defines all possible template-based extensions of O w.r.t. Λ

○ $\mathcal{S_V}$ is a *validating instantiation set* – an instantiation set that defines all minimal instantiations preserving inconsistency

In addition, we define:

○ $\langle O, \Lambda, \mathcal{S_G}, \mathcal{S_V} \rangle$ is *canonical* if $\mathcal{S_G}$ is closed under inclusion.

○ $\mathcal{S_C} = \{\Sigma_G \in \mathcal{S_G} \mid \forall \Sigma_V \in \mathcal{S_V} : \Sigma_V \nsubseteq \Sigma_G\}$ is a *combined instantiation set* w.r.t. some evolution schema $\langle O, \Lambda, \mathcal{S_G}, \mathcal{S_V} \rangle$.

An example of a canonical evolution schema $\langle O, \Lambda, \mathcal{S_G}, \mathcal{S_V} \rangle$ is visualized in Figure 3.2. The generating instantiation set $\mathcal{S_G}$ is visualized by the color green, while the validating instantiation set $\mathcal{S_V}$ is visualized by the color red. Note that, for any graph of instantiation set closed under inclusion (such as $\mathcal{S_G}$) it holds that if it contains a node representing substitution set Σ it also contains a complete sub-graph w.r.t. Σ.

3.2.2 $\mathcal{ALC}_x$ Semantics

Similarly to *MIPS* described in Section 2.1, but adapted to *ontology templates*, we say that an instantiation Σ is a *MIPS minimal inconsistency preserving set* w.r.t. an ontology O and an ontology template set Λ if the following conditions hold:

○ *inconsistency condition*, i.e. $(O \cup IF(\Sigma, \Lambda))$ is inconsistent

○ *minimality condition*, i.e. $\forall \Sigma' \subset \Sigma \implies (O \cup IF(\Sigma', \Lambda))$ is consistent

Let $ES = \langle O, \Lambda, \mathcal{S_G}, \mathcal{S_V} \rangle$ be an evolution schema and $\mathcal{S_C}$ be its combined instantiation set.

○ A validating instantiation set $\mathcal{S'_V}$ is *correct* w.r.t. evolution schema $\langle O, \Lambda, \mathcal{S_G}, \mathcal{S_V} \rangle$, if O is consistent and for each $\Sigma \in \mathcal{S'_V}$, such that $\Sigma \in cl(\mathcal{S_G}) : \Sigma$ is *MIPS* w.r.t. O and Λ.

○ The evolution schema ES is *(necessarily) consistent* if $\mathcal{S_V}$ is correct w.r.t. $\langle O, \Lambda, \mathcal{S_G}, \emptyset \rangle$ and for every substitution set $\Sigma \in \mathcal{S_C}$ there exists an interpretation $\mathcal{I}$ such that $\mathcal{I} \vDash (O \cup IF(\Sigma, \Lambda))$.

Recall the example of the evolution schema $\langle O, \Lambda, \mathcal{S_G}, \mathcal{S_V} \rangle$ from Figure 3.2 as well as the definitions of substitutions $\sigma_1, \sigma_2, \sigma_3, \sigma_4$. We will show that the evolution schema in the figure is consistent. First, it is easy to see that $\mathcal{S_V} = \{\{\sigma_2, \sigma_3\}, \{\sigma_2, \sigma_4\}\}$ is correct w.r.t. the evolution schema – ontologies $O \cup IF(\{\sigma_2, \sigma_3\}, \Lambda)$ and $O \cup IF(\{\sigma_2, \sigma_4\}, \Lambda)$ are both inconsistent, while ontologies O, $O \cup IF(\{\sigma_2\}, \Lambda)$, $O \cup IF(\{\sigma_3\}, \Lambda)$, and $O \cup IF(\{\sigma_4\})$ are consistent. Second, according to the provided definition $\mathcal{S_C} = \{\{\sigma_1, \sigma_3, \sigma_4\}, \{\sigma_1, \sigma_3\}, \{\sigma_1, \sigma_4\}, \{\sigma_3, \sigma_4\}, \{\sigma_1\}, \{\sigma_2\}, \{\sigma_3\}, \{\sigma_4\}, \{\}\}$. It can be shown that for each $\Sigma \in \mathcal{S_C}$ an ontology $O \cup IF(\Sigma, \Lambda)$ is consistent (i.e. has model). Figure 3.3 shows a combined instantiation set $\mathcal{S_C}$ as a blue sub-graph. The red sub-graph shows all inconsistent instantiations of the evolution. $\{\sigma_2, \sigma_3, \sigma_4\}$ is an example of inconsistent instantiation which is not minimal. $max_\subseteq(\mathcal{S_C})$ is a set of all nodes that does not have a parent in $\mathcal{S_C}$, i.e. $\{\{\sigma_1, \sigma_3, \sigma_4\}, \{\sigma_3, \sigma_4\}, \{\sigma_3\}\}$. Moreover, $\mathcal{S_V}$ is *MIPS* w.r.t. O and Λ and note, that this is true for each consistent evolution.
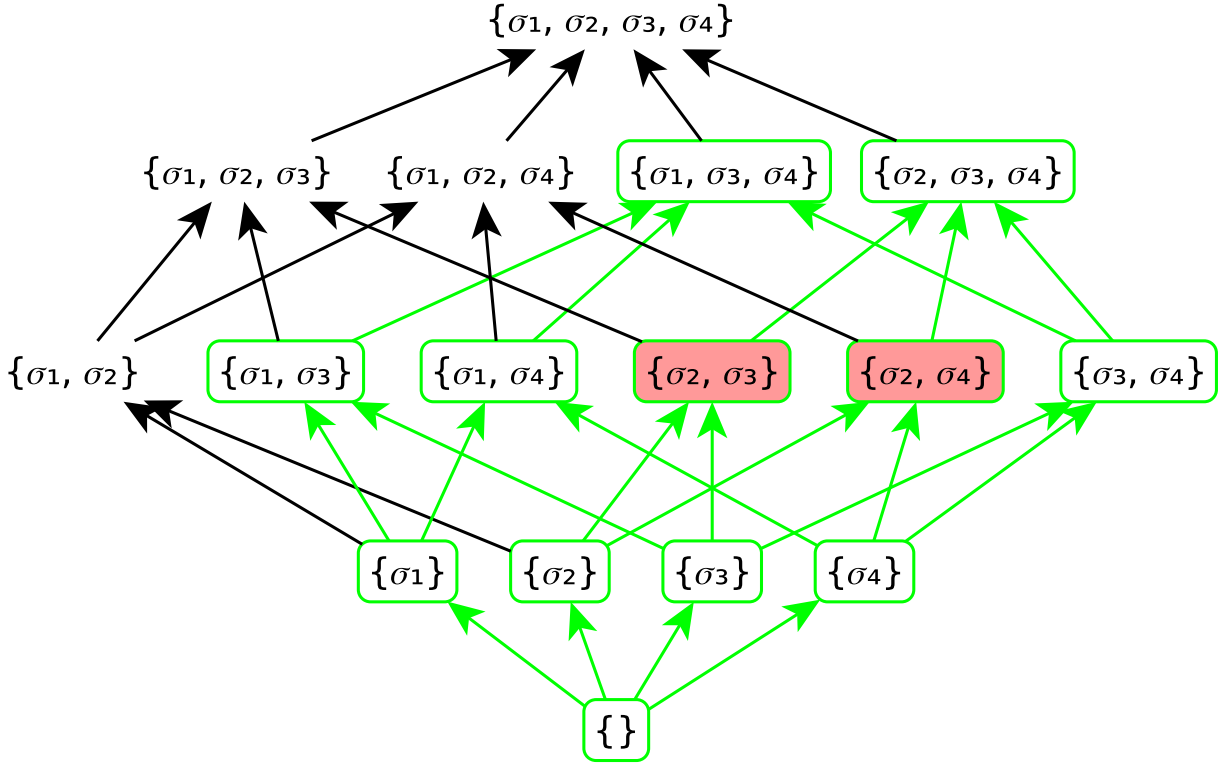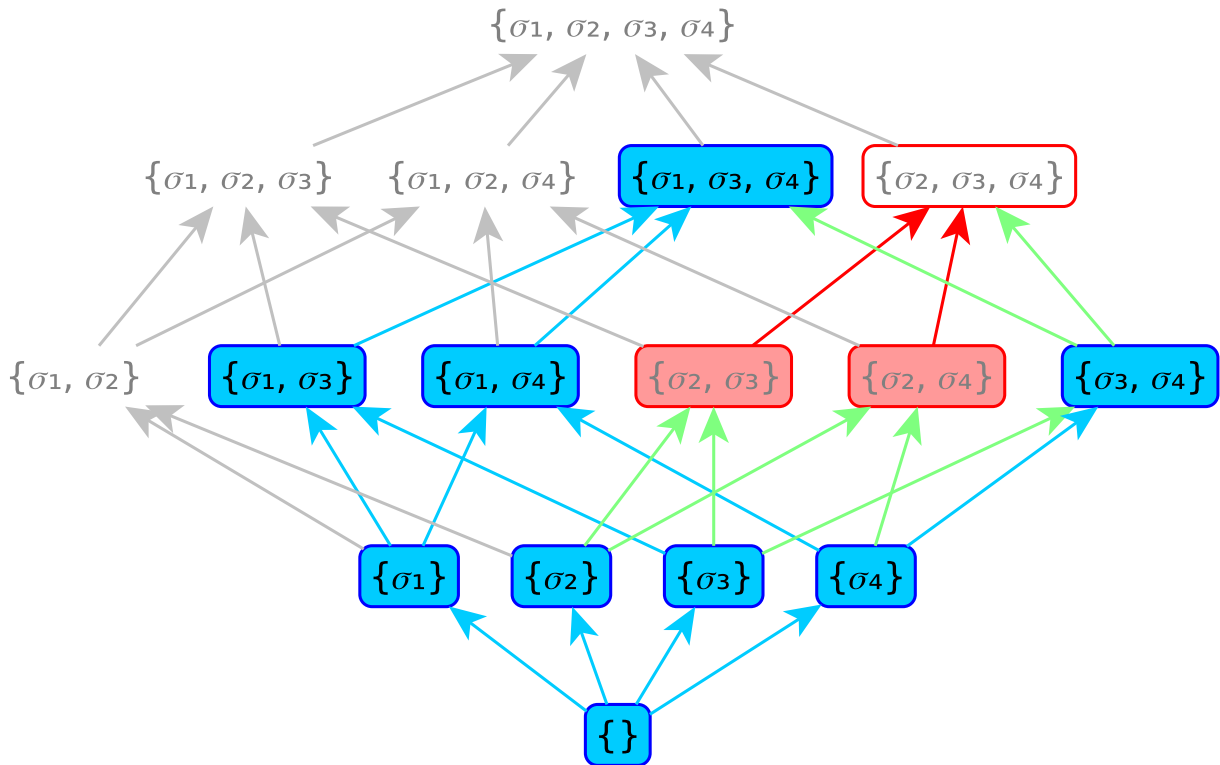
Figure 3.3: Complete modification graph w.r.t. a canonical evolution schema $\langle O, \Lambda, \mathcal{S}_\mathcal{G}, \mathcal{S}_\mathcal{V} \rangle$, sub-graph with red and blue nodes represents $\mathcal{S}_\mathcal{G}$, red sub-graph represents a set of inconsistent instantiations, blue sub-graph represents $\mathcal{S}_C$.

3.3 Reasoning Service

OWL2 and many other $\mathcal{DL}$ based languages provide a set of useful services for querying, validating and debugging errors within an ontology. One of the most basic services is consistency checking of an ontology. If the consistency check fails during the manual development of the ontology, an ontology engineer typically starts appropriate debugging services to find the root cause of the problem and fix it.

Let's imagine, that an ontology engineer identifies some higher level patterns within the ontology and creates an *ontology template* in order to simplify his work. He assigns a *domain* to each *variable* within the template which significantly reduces his time when filling in the templates. Unfortunately, he is still not satisfied as there are many dependent variables in the template. Some of them could be filled in automatically, others could have a set of values significantly reduced based on already assigned variables of the form. He decides to create a better instantiation function (not only based on variable domains) by providing some additional relationships between particular variables of the template. He creates new *ontology templates*, fills in some data based on them and runs a consistency check, which fails. After he spends quite a long time trying to find out the ontology error, he decides to add a *validation constraint* for

ontology templates, so it will not happen again in such a situation. This again reduces variable values within the forms of *ontology templates*. He also adds a non-technical textual description of *validation constraint* as he plans to provide some *ontology templates* forms to domain experts to populate the ontology. The only thing he is missing is: 1) a way to find out whether the *validation constraint* is not too restrictive, i.e. it does not exclude some consistent *ontology template groundings*, 2) whether there are some other possible inconsistences that *ontology templates* can create. Points 1) and 2) can be answered by new reasoning service for *ontology templates* as follows.

Reasoning services Let $\langle O, \Lambda, \mathcal{S}_{\mathcal{G}}, \mathcal{S}_{\mathcal{V}} \rangle$ be an ontology evolution scheme and $\mathcal{S}^{\prime}_{\mathcal{V}}$ an instantiation set:

- *validation constraint check* – takes $\mathcal{S}^{\prime}_{\mathcal{V}}$ and $\langle O, \Lambda, \mathcal{S}_{\mathcal{G}}, \mathcal{S}_{\mathcal{V}} \rangle$ as parameters and returns true if $\mathcal{S}^{\prime}_{\mathcal{V}}$ is a correct validation instantiation set w.r.t. evolution scheme $\langle O, \Lambda, \mathcal{S}_{\mathcal{G}}, \mathcal{S}_{\mathcal{V}} \rangle$. Otherwise it returns an ontology that proves the incorrectness of the $\mathcal{S}^{\prime}_{\mathcal{V}}$. Algorithm 1 provides a description of the service.

- *evolution scheme consistency check* – takes $\langle O, \Lambda, \mathcal{S}_{\mathcal{G}}, \mathcal{S}_{\mathcal{V}} \rangle$ as a parameter returns true if $\langle O, \Lambda, \mathcal{S}_{\mathcal{G}}, \mathcal{S}_{\mathcal{V}} \rangle$ is consistent. Otherwise it returns an ontology that proves either the inconsistency of the whole scheme or the incorrectness of the validation set $\mathcal{S}_{\mathcal{V}}$. Algorithm 3 provides a description of the service.

In the following text we define three algorithms needed for reasoning services together with proof of their soundness and completeness. The algorithms have two outputs – an output value and global variable E which represents an error by tuple of a textual message and ontology that proves the error.

Algorithm 1 checks the correctness of validation constraints w.r.t. some evolution schema.

- *Termination* – The only looping structures are "foreach" constructs over a finite immutable set, thus the algorithm terminates.

- *Soundness* – If the algorithm returned *true*, thus the execution did not step inside the code within the lines *(2-4)*, *(7-9)* and *(12-14)* – we will make abbreviation *MISS()* to express this fact. The *MISS(2-4)* implies that the ontology O is consistent, the *MISS(7-9)* implies that for each $\Sigma \in (\mathcal{S}^{\prime}_{\mathcal{V}} \cap cl(\mathcal{S}_{\mathcal{G}}))$ the *inconsistency criterion* holds. The *MISS(12-14)* implies that $\forall \sigma \in \Sigma : O \cup IF(\Sigma - \{\sigma\}, \Lambda)$ is consistent. For an arbitrary $\Sigma' \subset \Sigma$ it is true that $\Sigma' \subseteq \Sigma - \{\sigma\}$ for some σ. Thus $(O \cup IF(\Sigma', \Lambda)) \subseteq (O \cup IF(\Sigma - \{\sigma\}, \Lambda))$ and the *minimality condition* implies from *monotonicity of entailment*[2] [1] in $\mathcal{ALC}$.

- *Completeness* – Let O be a consistent ontology, $\Sigma \in (\mathcal{S}^{\prime}_{\mathcal{V}} \cap cl(\mathcal{S}_{\mathcal{G}}))$ and Σ is MIPS w.r.t. O and Λ. The algorithm returns *true* as *MISS(2-4)*, *MISS(7-9)* and *MISS(12-14)* is trivially guaranteed from preconditions.

---

[2]Monotonic logic such as $\mathcal{ALC}$ ensures that if O is consistent, than any subset O' is also consistent.

○ *Complexity* – Let O, Λ, $\mathcal{S}$ be an ontology, an ontology template set and an instantion set, respectively. We will use $l(O)$, $l(\Lambda)$, $l(\mathcal{S})$ to denote the length (size) of their encoding within the algorithm. The upper bound complexity of the ontology consistency (satisfiability) check of $\mathcal{ALC}$ is PSPACE [1]. Thus lines (2-4) are in PSPACE w.r.t. $l(O)$. The computation of the condition on line (5) is at most polynomial w.r.t. $\mathcal{S}_\mathcal{G}$ and $\mathcal{S}'_\mathcal{V}$. The size of O' and O'' is at most $l(O) + l(\Lambda) * l(\mathcal{S}'_\mathcal{V})$, thus polynomially bounded by $l(O)$, $l(\Lambda)$, and $l(\mathcal{S}'_\mathcal{V})$. Loop (5-14) is executed at most $l(\mathcal{S}'_\mathcal{V})$ times and its inner loop is thus executed at most $l(\mathcal{S}'_\mathcal{V})^2$ times. The size of O'' and O' is at most $l(O) + l(\Lambda) * l(\mathcal{S}'_\mathcal{V})$. Thus the number of consistency checks within lines (5-14) as well as the size of the checked ontology is bounded polynomially and therefore can be computed in PSPACE. From the above analysis it implies that Algorithm 1 uses input $l(\mathcal{S}_\mathcal{G})$ to compute only line (5) which can be done polynomially w.r.t. $l(\mathcal{S}_\mathcal{G})$ and $l(\mathcal{S}'_\mathcal{V})$. The algorithm is in PSPACE w.r.t. $l(O)$, $l(\Lambda)$, $l(\mathcal{S}'_\mathcal{V})$ and $l(\mathcal{S}_\mathcal{G})$.

Algorithm 1: Check validating constraint correctness

---

1 **is_correct_VC** $(\mathcal{S}'_\mathcal{V}, \langle O, \Lambda, \mathcal{S}_\mathcal{G}, \mathcal{S}_\mathcal{V} \rangle)$: *boolean* **is**

    Result: Returns true, if $\mathcal{S}'_\mathcal{V}$ is a correct validation instantiation set w.r.t. the evolution scheme $\langle O, \Lambda, \mathcal{S}_\mathcal{G}, \mathcal{S}_\mathcal{V} \rangle$. Otherwise it returns false and an ontology that proves incorrectness of the $\mathcal{S}'_\mathcal{V}$.

2   **if** *not* `is_consistent(`O`)` **then**

3         $E \leftarrow \langle$"core ontology is not consistent"$, O \rangle$

4   **return** *false*

5   **foreach** $\Sigma \in (\mathcal{S}'_\mathcal{V} \cap cl(\mathcal{S}_\mathcal{G}))$ **do**

6         $O' \leftarrow (O \cup IF(\Sigma, \Lambda))$

7   **if** `is_consistent(`O'`)` **then**

8             $E \leftarrow \langle$"validation constraint not correct"$, O' \rangle$

9   **return** *false*

10   **foreach** $\sigma \in \Sigma$ **do**

11             $O'' \leftarrow (O \cup IF(\Sigma - \{\sigma\}, \Lambda))$

12   **if** *not* `is_consistent(`O''`)` **then**

13                 $E \leftarrow \langle$"validation constraint is not minimal"$, O'' \rangle$

14   **return** *false*

15   **return** *true*

Algorithm 2 computes an instantiation Σ' which is *MIPS* w.r.t. an ontology O, an ontology template set Λ, and an instantiation Σ. It assumes to call two external functions: compute single MIPS of an ontology and compute a hitting set, both explained in section 2.1. The algorithm works as follows:

Algorithm 2: Get single *MIPS* instantiation

---

1 **get_single_MIPS_instantiation** $(O, \Lambda, \Sigma) : \Sigma'$ **is**

> **Result**: Returns an arbitrary instantiation $\Sigma' \subseteq \Sigma$ that causes inconsistency w.r.t. O and is minimal with respect to template-based extension of O using Λ and Σ. In the case where O is inconsistent or instantiation Σ does not lead to inconsistence, it returns $\emptyset$

2   **if** *not* is_consistent(O) **then**

3      $E \leftarrow \langle$ "core ontology is not consistent", $O \rangle$

4   **return** $\emptyset$

5    $M \leftarrow \{\}$

6    $O'' \leftarrow O$

7   **foreach** $\sigma \in \Sigma$ **do**

8      $O' \leftarrow IF(\sigma, \Lambda) - O$

9      $M \leftarrow M \cup \{\langle \alpha, \sigma \rangle \mid \alpha \in O'\}$

10      $O'' \leftarrow O'' \cup O'$

11    $O_{sMIPS} \leftarrow$ get_single_MIPS(O'')

12    $\mathcal{S}_{MIPS} \leftarrow \{\{\sigma \mid \langle \alpha, \sigma \rangle \in M\} \mid \alpha \in O_{sMIPS}\}$

13   **return** get_single_minimal_HTS($\mathcal{S}_{MIPS}$)

○ lines *(5-10)* define ontology $O'' = IF(\Sigma, \Lambda)$ and relation M that maps each axiom α such that $\alpha \notin O$ to a set of substitutions $\{\sigma_i\}$ that could lead to the grounded axiom α. σ_i represents a ground substitution of an ontology template from Λ,

○ lines *(11-12)* compute single *MIPS* of ontology $O \cup IF(\Sigma, \Lambda)$ i.e. a set of axioms O_{sMIPS} which are transformed to a set of substitution sets $\mathcal{S}_{sMIPS}$ where each substitution represents an axiom from O_{sMIPS},

○ line *(13)* computes minimal hitting set of $\mathcal{S}_{MIPS}$.

To prove the correctness of the algorithm let us assume that O is consistent, $O \cup IF(\Sigma, \Lambda)$ is inconsistent, but the algorithm outputs Σ' failing the *inconsistency condition*. From the monotonicity of entailment and the fact that $O_{sMIPS} \subseteq O \cup IF(\Sigma, \Lambda)$ it implies that O_{sMIPS} must be a consistent ontology which is a contradiction. Let us assume that Σ' satisfies the *inconsistency condition* but fails the *minimality condition*. Thus there exists $\sigma \in \Sigma'$ and $\Sigma'' = \Sigma' - \{\sigma\}$ such that $O \cup IF(\Sigma'', \Lambda)$ is inconsistent. This implies that either the computed hitting set is not minimal or the axiom set returned by the single mips algorithm is not minimal – this is a contradiction, and thus we proved that Σ' must satisfy both the *inconsistence* and *minimality* conditions. The *termination* of the algorithm is satisfied because only looping structures are "foreach" constructs over a finite immutable set.

The *complexity* of Algorithm 2 can be evaluated using function l that defines the size of the encoding of the input as explained above. Lines (2-4) can be evaluated in PSPACE w.r.t. l(O).

The size of O'' is polynomial w.r.t. $l(O)$, $l(\Lambda)$ and $l(\Sigma)$. The algorithm for single MIPS is PSPACE [13] and the algorithm for minimal HTS is NP-complete [15] problem. Thus Algorithm 2 is PSPACE w.r.t. $l(O)$, $l(\Lambda)$, and $l(\Sigma)$.

Algorithm 3: Check evolution schema consistency

---

1 **is_consistent_ES** $(\langle O, \Lambda, \mathcal{S}_\mathcal{G}, \mathcal{S}_\mathcal{V} \rangle)$: *boolean* **is**

    Result: Returns true, if evolution schema $\langle O, \Lambda, \mathcal{S}_\mathcal{G}, \mathcal{S}_\mathcal{V} \rangle$ is consistent. Otherwise it returns false and an ontology that proves its inconsistency.

2   **if** *not* `is_correct_VC`$(\mathcal{S}_\mathcal{V}, \langle O, \Lambda, \mathcal{S}_\mathcal{G}, \emptyset \rangle)$ **then**

3   **return** *false*

4     $\mathcal{S}_C \leftarrow \{\Sigma_\mathcal{G} \in \mathcal{S}_\mathcal{G} \mid \forall \Sigma_\mathcal{V} \in \mathcal{S}_\mathcal{V} : \Sigma_\mathcal{V} \nsubseteq \Sigma_\mathcal{G}\}$

5   **foreach** $\Sigma \in max_\subseteq(\mathcal{S}_C)$ **do**

6         $\Sigma_{sMIPS} \leftarrow$ `get_single_MIPS_instantiation`(O, Λ, Σ)

7   **if** $\Sigma_{sMIPS} \neq \emptyset$ **then**

8             $E \leftarrow \langle$ "evolution generates inconsistent ontology", $O \cup IF(\Sigma_{sMIPS}, \Lambda) \rangle$

9   **return** *false*

10   **return** *true*

Algorithm 3 checks the consistency of the evolution schema. It uses an algorithm to compute single *MIPS* instantiation only to return a meaningful explanation if the consistency check fails.

- ○ *Termination* – implies from only one "foreach" constructs over a finite set.

- ○ *Soundness* – Let us assume the algorithm returned *true*. Thus from *MISS(2-3)* it implies that $\mathcal{S}_\mathcal{V}$ is correct and O is consistent. From lines *(5-9)* and *MISS(7-9)* it implies that for each $\Sigma \in max_\subseteq(\mathcal{S}_C)$ it implies that $O \cup IF(\Sigma, \Lambda)$ is consistent. For any $\Sigma' \in \mathcal{S}_C$ it holds that is subset of Σ and again due to monotonicity it implies that $O \cup IF(\Sigma', \Lambda)$ is consistent, thus there exists an interpretation $\mathcal{I}$ that satisfies the ontology.

- ○ *Correctness* – This assumes we have interpretation $\mathcal{I}$ for each $\Sigma \in \mathcal{S}_C$ and that lines *(7-9)* are not executed. Thus the algorithm returns *true*.

- ○ *Complexity* – Let l be a function defining the size of the encoding of the input as used above. The upper bound complexity of lines (2-3) is PSPACE w.r.t. $l(\mathcal{S}_\mathcal{V})$, $l(O)$, and $l(\Lambda)$. The computation of $\mathcal{S}_\mathcal{G}$ at line (4) is polynomial w.r.t. $l(\mathcal{S}_\mathcal{G})$ and $l(\mathcal{S}_\mathcal{V})$. Loop (5-9) is evaluated at most $l(\mathcal{S}_\mathcal{G})$ times. Single MIPS instantiations (line 6) is computed in PSPACE w.r.t. $l(O)$, $l(\Lambda)$, $l(\mathcal{S}_\mathcal{G})$) as explained above. Thus Algorithm 3 is in PSPACE w.r.t. $l(O)$, $l(\Lambda)$, $l(\mathcal{S}_\mathcal{G})$, and $l(\mathcal{S}_\mathcal{V})$.

3.3.1 Alternative approach for evolution schema consistency

Until now the algorithms for checking the correctness of validating constraints (Algorithm 1) and consistency evolution (Algorithm 3) only relied on a standard reasoning service for checking the

consistency of an ontology. Although the reasoning service to compute single MIPS is used by Algorithm 2, it is there only to provide a more concise explanation of an error to a user.

If the following text we will introduce an alternative approach to check the consistency of an evolution schema. However, this algorithm will directly depend on a more advanced service of a reasoner, i.e. the computation of all MIPS of an ontology. The algorithm is straightforward: (1) it computes all MIPS of an ontology that consist of a core ontology and all possible groundings of all ontology templates, (2) translates MIPS represented by ontological axioms into MIPS represented by instantiations, (3) checks whether the combined instantiation set of the evolution schema is compliant with computed MIPS of instantiations.

The computation of all MIPS and their translation from a set of ontological axioms to a set of instantiations is described in Algorithm 4. Compared to Algorithm 2 for computation of single MIPS, the algorithm requires multiple calls of the procedure to find all minimal hitting sets.

The alternative algorithm for checking the consistency of an evolution schema is described in Algorithm 5. Termination, soundness and correctness can be shown same way as the previous algorithm. The complexity is in addition limited by computation of MIPS of the ontology which is NP-complete problem.

Algorithm 4: Get all *MIPS* instantiations

1 **get_all_MIPS_instantiations** $(O, \Lambda, \Sigma) : \mathcal{S}$ **is**

  Result: Returns a set of all instantiations $\Sigma' \subseteq \Sigma$ that causes inconsistency w.r.t. O and is minimal with respect to template-based extension of O using Λ and Σ. In the case where O is inconsistent or instantiation Σ does not lead to inconsistence, it returns $\emptyset$

2 | **if** *not* is_consistent(O) **then**

3 | | $E \leftarrow \langle$"core ontology is not consistent", $O\rangle$

4 | | **return** $\emptyset$

5 | $M \leftarrow \{\}$

6 | $O'' \leftarrow O$

7 | **foreach** $\sigma \in \Sigma$ **do**

8 | | $O' \leftarrow IF(\sigma, \Lambda) - O$

9 | | $M \leftarrow M \cup \{\langle\alpha, \sigma\rangle \mid \alpha \in O'\}$

10 | | $O'' \leftarrow O'' \cup O'$

11 | $\mathcal{S} \leftarrow \emptyset$

12 | **foreach** $O_{sMIPS} \in$ get_all_MIPS(O'') **do**

13 | | $\mathcal{S}_{MIPS} \leftarrow \{\{\sigma \mid \langle\alpha, \sigma\rangle \in M\} \mid \alpha \in O_{sMIPS}\}$

14 | | $\mathcal{S} \leftarrow \mathcal{S} \cup$ get_all_minimal_HTS($\mathcal{S}_{MIPS}$)

15 | **return** $\mathcal{S}$

Algorithm 5: Check evolution schema consistency alternative

---

1 is_consistent_ES ($\langle O, \Lambda, \mathcal{S}_{\mathcal{G}}, \mathcal{S}_{\mathcal{V}} \rangle$) : *boolean* **is**

    Result: Returns true, if evolution schema $\langle O, \Lambda, \mathcal{S}_{\mathcal{G}}, \mathcal{S}_{\mathcal{V}} \rangle$ is consistent. Otherwise it returns false and an ontology that proves its inconsistency.

2   **if** *not* `is_correct_VC`($\mathcal{S}_{\mathcal{V}}, \langle O, \Lambda, \mathcal{S}_{\mathcal{G}}, \emptyset \rangle$) **then**

3   **return** *false*

4 $\mathcal{S}_C \leftarrow \{\Sigma_{\mathcal{G}} \in \mathcal{S}_{\mathcal{G}} \mid \forall \Sigma_{\mathcal{V}} \in \mathcal{S}_{\mathcal{V}} : \Sigma_{\mathcal{V}} \nsubseteq \Sigma_{\mathcal{G}}\}$

5 $\Sigma_C \leftarrow \bigcup_{\Sigma \in \mathcal{S}_C} \Sigma$

6 $\mathcal{S}_{MIPS} \leftarrow$ `get_all_MIPS_instantiations`(O, Λ, Σ_C)

7 $\Sigma_{sMIPS} \leftarrow$ `get_any_element`($\mathcal{S}_C \cup \mathcal{S}_{MIPS}$)

8   **if** $\Sigma_{sMIPS} \neq \emptyset$ **then**

9 $E \leftarrow \langle$"evolution generates inconsistent ontology", $O \cup IF(\Sigma_{sMIPS}, \Lambda)\rangle$

10   **return** *false*

11   **return** *true*

3.3.2 Notes on algorithms for consistency of evolution schema

Note that computation of maximal non-conflicting set in Algorithm 3 (i.e. line 5) is in fact the hitting set problem, which as will be shown in Section 4.4, is the performance bottleneck of the approach. On the other hand, Algorithm 5 computation of MIPS is very similar problem. The difference is that Algorithm 3 computes it on conflicting sets of instantiations, while Algorithm 5 computes it at level of ontological axioms. Moreover, it is easy to see that both algorithms can be implemented incrementally (i.e. checking one maximal ontology/diagnosis of conflict at a time). In Algorithm 5 it is possible due to incremental support for generation of justifications of the inconsistency, which is implemented in existing reasoners (e.g. Pellet [53]). Similarly, for Algorithm 3, it is possible to generate one minimal diagnosis at a time with an hitting set algorithm such as Reiter's algorithm.

3.4 Methodology for Defining Ontology Evolution Scenarios

The reasoning service as defined in this chapter provides a useful tool to define and discover ontology evolution scenarios. However the complexity of those algorithms shows their limitations as well. This section proposes a methodology to define ontology evolution scenarios with respect to those limitations in general. A more detailed discussion on limitations and related suggestions for improvement will be described in Section 4.4 which provides an evaluation of the implemented prototype.

3.4.1 Intended uses of evolution scenarios

Based on our experience with the above formalization of ontology templates we identified the following situations where ontology templates are useful:

- **Direct interaction of the domain expert with the ontology** – this is our main use of the formalization which was tested in the MONDIS project. In this situation, our framework is used to explain some consequences of his actions to the domain expert. Each validation constraint describes a set of similar explanations parameterized by concrete values that form the instantiations of the ontology templates. In such situations reasoning services as described above are crucial to use it effectively.

- **Model-driven applications** – this is similar to the previous situation with the difference that the scope of the application is typically too big to be included in one scenario due to the computational limitations of the reasoning service. In such situations we propose to define a full scenario that can be either directly integrated into the application as show in Chapter 5 or just be used to keep track of all instantiations. The complete scenario should be restricted to a new "examining" scenario with a smaller problem space that would be feasible for experiments with the reasoning service (see Section 4.4).

- **Auxiliary tool for ontological experiments** – in addition to other standard services available for ontologies (such as consistency check, classification, justifications etc.), the reasoning service brings new opportunities for an ontology developer. It allows the checking of certain implications of the ontology with higher order statements, which is one of the motivations originating from the privacy protection use-case (see Section 5.2).

- **Management of higher level statements** – Within the MONDIS project it turned out to be very useful to organize ontology templates into evolution scenarios even if they are not used with the reasoning service. Keeping track of higher level statements of the ontology is useful when (1) knowledge is gathered from external resources and needs to be transformed to an ontology (see Section 5.1), (2) knowledge needs to be transformed from one ontology design pattern to another, (3) the higher level statements are complex and manually entered by ontology editor. An enhancement of an ontology editor in such a case is described in Section **??**. In situations with many scenarios and ontology templates we suggest to organize scenarios into purpose specific taxonomies while organizing ontology templates into more generic categories such as the organization of ontology design patterns.

3.4.2 Main workflow for defining evolution scenarios

The formalization of the ontology evolution scenario can be decoupled into a sequence of multiple tasks. We will provide a guidelines for each task separately and demonstrate it on our motivating example of building components and materials from Chapter 1. The workflow of the defining ontology evolution scenario can be formulated info the following tasks:

- ○ **Identify the purpose of the evolution scenario** – The purpose of the scenario defines its intended goal, i.e. the main function for which it will be used. As mentioned in Chapter 1 the goal of the controlled evolution scenario is to extend ontology in terms of axioms.

  Example: "Enriching of the MONDIS specific part of Monument Damage Ontology with relations between types of Building components and materials."

- ○ **Identify end-users and intended uses of the evolution scenario** – End-users are users that will be interacting with the evolution scenario. This task specifies them together with their intended uses of the scenario as specified above.

  Example: "The scenario will be used in the mind-mapping software Ontomind [54], where domain experts should have access to the ontological consequences of their actions immediately."

- ○ **Identify the scope of the evolution scenario** – The scope of the evolution scenario includes the definition of the *core ontology* and *scenario context*. The core ontology is used as the starting point of the evolution scenario and the scenario context is a source of input data for the definition of atomic operations within the scenario. They both must be selected carefully with respect to intended uses (see Section 3.4.1) as they define the problem space of algorithms for computing consequences of actions within the evolution scenario. To reduce the size of the core ontology one can exclude axioms that will not participate in the inconsistency of the evolved ontology. This can be done by the extraction of the semantic module of the ontology [55]. Moreover, if intended end-users are domain experts, some of the ontological consequences heavily depending on the core ontology might be hard to explain to them, if they did not participate in the formalization of the core-ontology. Another option to reduce the problem space is to run the scenario at the beginning with a smaller core ontology and later, when some atomic operations are already applied to the evolution (i.e. the space of possible evolutions is restricted), we can compute the restricted evaluation scenario again. Although the scenario context is the same as the core ontology in many cases, in general it can be any artifact from which atomic additions are generated, such as a third-party ontology, a file in CSV format, a Microsoft Excel Spreadsheet document, or a mindmap. A general rule to reduce the problem space using the scenario context is to choose a context that will generate minimum atomic operations of the evolution that is required. For the implementation specific analysis of the problems see Section 4.4.

  Example: "As a core ontology and scenario context for our scenario we will use the core conceptual model of Monument Damage Ontology extended with simple taxonomic relations of the form $A_1 \sqsubseteq A_2$, where A_1 and A_2 are atomic concepts." In this case, the evolution scenario is very simple to explain to the user. Later when some of the relations are already asserted we can add other types of axioms (such as disjointness between atomic classes) or even a full semantic sub-module related to the evolution scenario.

- ○ **Define atomic operations and ontology templates** – The purpose of this task is to define all possible atomic additions that can be applied to the core ontology and their related ontology templates from which they will be generated. This is defined by the purpose of

the evolution scenario and typically requires the study of ontology design patterns to define proper ways to formalize ontology templates. The distinguishing property of ontology templates in comparison to a design patterns is that their identity is also defined by the way their instantiations are computed from a scenario context. This allows the reuse of the ontology templates in more a fine-grained way than ontology design patterns are typically used. We suggest using this feature of ontology templates if managing multiple ontology scenarios.

Example: "We will use ontology templates $T_1 = \{\$COMP_1 \sqsubseteq \forall hasMaterial.\neg\$MAT_1 \}$ and $T_2 = \{\_c : \$COMP_2 \sqcap \exists hasMaterial.\$MAT_2 \}$, where values for building components and materials will be from the taxonomy of the class *Component* and class *Material* respectively."

○ **Define generating constraints** – Generating constraints are restrictions on the usage of atomic additions that are not encoded in the *core ontology*. Within semantic web applications they can be encoded by SPARQL or SWRL. Generally, any query/rule language with closed world interpretation can be used. In order to maintain them effectively an implementation of ontology templates should directly incorporate them into the generation of instantiations as shown in Chapter 4.

○ **Define and check validating constraints** – Validating constraints are restrictions on the usage of atomic additions that are encoded in the *core ontology*. As a consequence they are in many cases not easy to find or to formulate explicitly. The provided reasoning service together with the help of an ontology reasoner supporting the generation of justifications of an entailment should be of great help in such cases. The reasoning service is limited only to checking consistency of the evolved ontology. On the other hand, due to the reducibility of entailment problems [13] it is easy to use this service for any entailment of the evolved ontology. Section 4.4 demonstrates how this reasoning service can be used to check the satisfiability of all classes. Workflow of actions to check correctness of validating constraints is described in Figure 3.4.

○ **Validate evolution scenario** – An ontology evolution scenario is consistent if its validation constraints explain "exactly" all restrictions that are encoded in the ontology, but not excluded by generating constraints. The importance of this task is very dependent on the intended use of the scenario as defined in Section 3.4.1. Workflow of actions to check consistency of evolution schema is described in Figure 3.4.
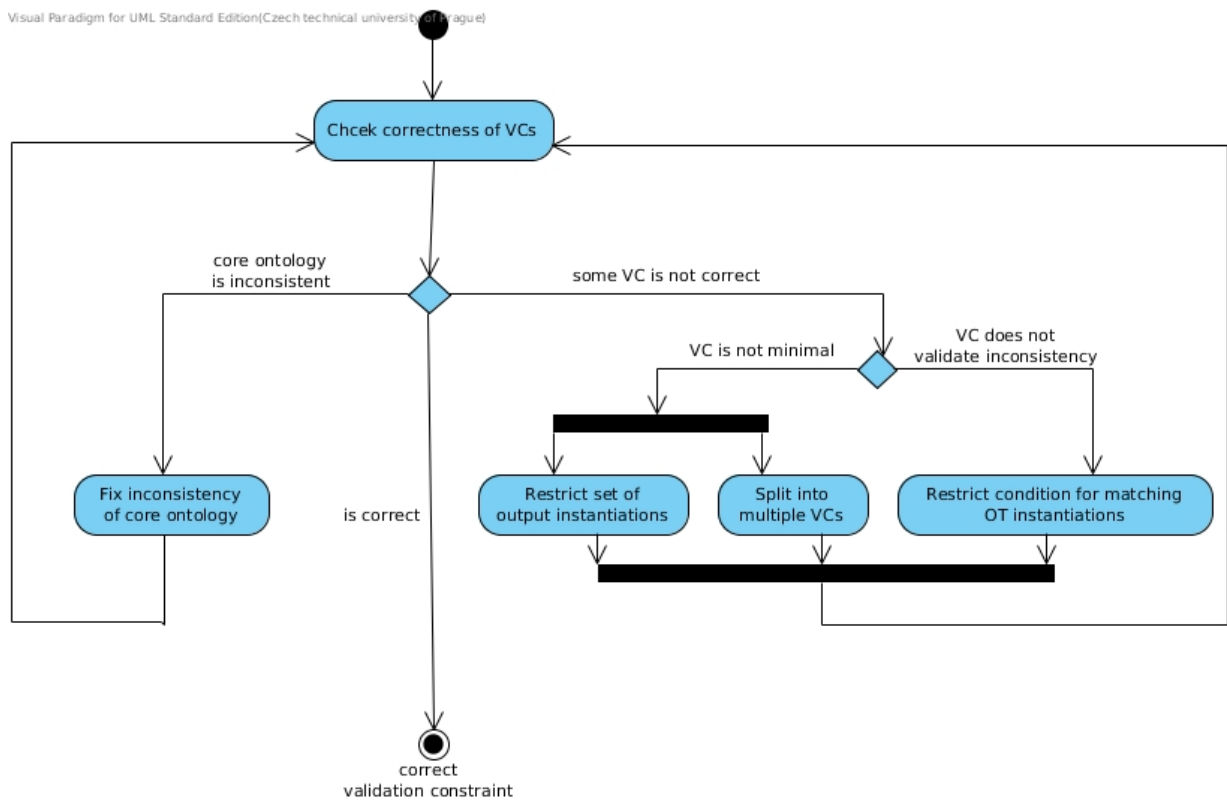
Figure 3.4: Workflow of actions for checking the correctness of validation constraints.
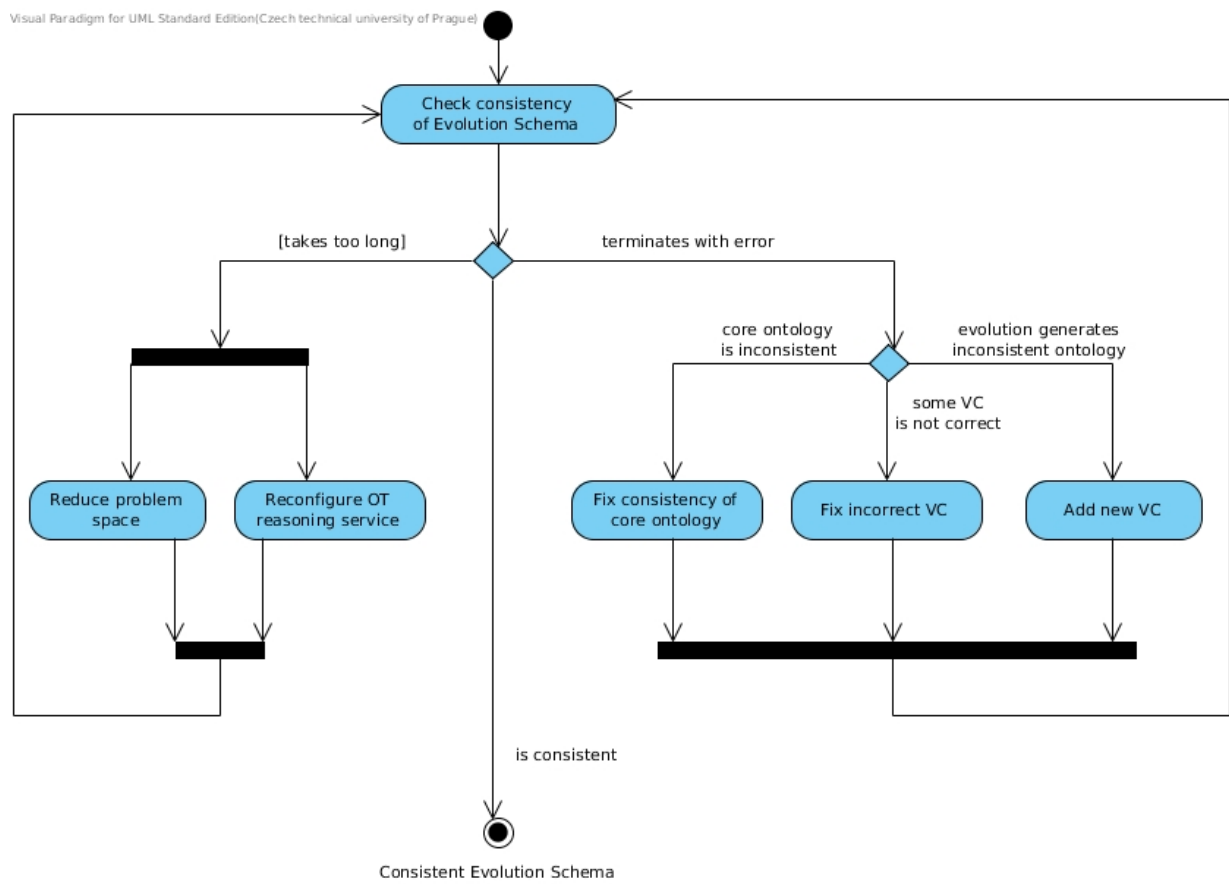
Figure 3.5: Workflow of actions for checking the consistency of evolution schema.

4 Software design and implementation

This chapter presents design and implementation of the formal framework presented in Chapter 3 and its related tools. First section describes a semantic web ontology to represent the conceptual model of the controlled evolution, which is enable to fulfill all scenarios discussed in the chapter. Second section describes design and implementation of command-line utilities for management of template-based evolution. Last two sections describes design and implementation of template-based reasoning service and its evaluation.

4.1 Ontology of SPARQL-based Controlled Evolution

Section 3.1 introduced conceptual model for scenarios of controlled ontology evolution. We will extend this model towards concrete representation of ontology templates, constraint and its instantiations and implement it in form of semantic web ontology that uses extended SPARQL query language encoded by SPIN vocabulary.

Purpose and Scope   Purpose of the Ontology of Controlled Evolution is to provide a knowledge model for description of scenarios and reasoning over controlled ontology evolution. The ontology should be able to represent complete scenario in terms of inputs and outputs of reasoning services within the controlled evolution.

Intended Uses and End-Users   Intended end-users are ontology engineers that can use use it for (1) specification of ontology-guided evolution scenarios for domain experts, (2) definition of atomic operations and restrictions of their use within model-driven systems, (3) exploring consequences of an evolution scenario, (4) management of higher level statements.

Non-functional requirements   The ontology should be divided into two separate parts providing (1) a consensual knowledge model of controlled ontology evolution and (2) concrete representations of inputs and outputs of reasoning services.

Functional requirements   We represent functional requirements in form of competency questions [48] that ontology should be able to answer.

(**CQ_1**) What are the ontology templates of this evolution scenario ?

(**CQ_2**) What is the core ontology that this ontology evolution extends ?

(**CQ_3**) What is the scenario context of this ontology evolution ?

(**CQ_4**) Which groundings of ontology templates are added in evolved ontology ?

(**CQ_5**) What validation/generation constraint are applicable to this ontology evolution ?

(**CQ_6**) How many groundings has this ontology template within this evolution scenario ?

(**CQ_7**) Which generation constraints are violated in evolved ontology ?

(**CQ_8**) Which validation constraints are violated in evolved ontology ?

(**CQ_9**) What are possible groundings that can be added to evolved ontology without breaking its consistency ?

(**CQ_10**) Which are minimal sets of groundings explaining inconsistency in evolved ontology ?

(**CQ_11**) What groundings i need to remove in order for the evolved ontology to become consistent ?

(**CQ_12**) How many violations of constrains are present in evolved ontology ?

(**CQ_13**) Which ontology templates cannot participate in any inconsistency of evolved ontology ?

(**CQ_14**) Which ontology templates can cause inconsistency of evolved ontology without participation of other ontology templates ?

(**CQ_15**) Are there any ontology templates that are restrictions this template ?

(**CQ_16**) What are variables of this ontology template ?

(**CQ_17**) What is variable assignment of this ontology template grounding ?

(**CQ_18**) How can be template groundings constructed from ontology template based on this evolution context ?

(**CQ_19**) Are there any templates that generate same groundings within this evolution scenario ?

Note that competency questions **CQ_1-15** can be answered from conceptual model from Section 3.1 and will be implemented by an ontology to which we will refer as "Ontology of Controlled Evolution". Requirements specified by **CQ_16-19** represents extension of this conceptual model that will be implemented as its extension to which we will refer as "Ontology of ARQ-based Controlled Evolution".

4.1.1 Ontology of Controlled Evolution

The conceptual model from Section 3.1 is in a straightforward way represented as OWL2 compliant ontology. Concepts are represented as OWL classes, while relations are represented as OWL object properties. Existential dependence between concepts is expressed in the ontology by existential quantification restrictions. As an example consider definition of ontology template instantiation : *OTInstantiation* subClassOf (*definesVariableAssignmentOf* some *OntologyTemplate*) and (*isCreatedIn* some *OntologyEvolutionScenario*).

4.1.2 SPARQL-based Extension

Ontology of SPARQL-based Controlled Evolution is an RDF ontology that imports Ontolog of Controlled Evolution. It uses SPIN vocabulary (see Section 2.1.5) to represent ontology templates as well as generating and validating constraints in terms of SPARQL queries[1]. Variables in ontology templates correspond to subset of output variables of SPARQL queries. Moreover due to RDF representation of SPARQL queries, we can reference and share parts of the SPARQL query such as graph patterns, variables etc. Examples of representation of ontology templates and constraints are provided in the following sections.

An alternative approach that was considered together with SPIN representation of ontology templates is OPPL scripting language. In comparison to SPIN, it is designed specificaly to query with OWL semantics and make operations over OWL ontologies. Therefore similar construct as we needed to represent in our ontology could be represented by OPPL in easier and more straightforward way. On the other hand, OPPL does not allow reuse of parts of the queries to such granularity as SPIN and is limited only to OWL2 compliant ontologies. The reuse of parts of the query, typically encoded using SPIN templates is key feature to manage ontology templates effectively. In following text we disscuss specific features and proposed used of the ontology separately.

Representation of ontology templates   The reasoning service for controlled ontology evolution from Section 3.2 allows for only variables in place of concepts. Our implementation extend this notion where variables can represent also roles and individuals of the ontology. This is useful for cases where the reasoning within the scenario is not required. Also note that the mentioned reasoning service can be easily extended to support such variables. In addition it allows to represent multi-valued variables in cases where ontology templates depends on dynamic number of variables. This occurs axioms using OWL2 union and intersection class expressions. Although SPARQL has very limited support for construction of such expressions from variable bindings, the SPIN vocabulary provides the extension function *tops:constructRDFList* to support it. It can be used in graph pattern matching part of the query to generate and bind specific parts of the RDF list into SPARQL variables. An example of SPARQL-based ontology template is shown in A.

---

[1]As an alternative we provide way to repesent the scenario by SPARQL textual form only, which has however limitations in reuse and some other advanced features of the ontology that are discussed later.

Referencing ontology templates from a domain ontology   We propose to reference representations of ontology templates from within domain ontologies using SPIN property *spin:imports*. It can be used to link the base URI of a domain ontology with a SPIN file, specified by the base URI of the SPIN file. For a SPIN constraint checker (or rule engine), the *spin:imports* keyword has the same meaning as *owl:imports*, i.e. all triples from the imported file will be added to the current RDF graph. However, the triples specified by *spin:imports* will not be imported in an OWL sense and therefore remain invisible to any OWL tool.

Representation of generating and validating constraints   Generating and validating constraints are defined by SPARQL select queries. They execute over the ontological model of the evolution scenario, core ontology and scenario context to return all combinations of ontology instantiations that should be excluded from the ontology evolution. Examples of generating and validating constrants are shown in 4.3. Note that this is possible only because ontology template groundings are represented in Ontology of Controlled Evolution as first class entities – OWL individuals.

4.2   Command-line Utilities for Management of Ontology Templates

We implemented platform-independent command-line interface for management of ontology templates. It supports templates as described in previous section as well as simple abstract templates represented in extended Manchester OWL syntax.

The command-line interface has following modules :

○ **find-axiom** – used to find set of axioms that were created from specified ontology template. Alternative finds set of axioms that are annotated in specified way.

○ **add-axiom** – used to add set of axioms into an ontology.

○ **remove-axiom** – used to remove set of axioms from an ontology.

○ **instantiate-template** – used to generated groundings of ontology templates that are optionally annotated by some meta-data using OWL2 annotation properties.

○ **check-validating-constraint** – provides interface for checking correctness of validating constraints w.r.t. some evolution schema as defined by Algorithm 1.

○ **check-evolution-schema** – provides interface for checking consistency of evolution schema as defined by Algorithm 3 and Algorithm 5.

○ **generate-evolution-scenario** – generates full representation of evolution scenario in Ontology of SPARQL-based Controlled Evolution that can be used as input for a semantic

web application. Optionally it can generates axioms and SPIN rules that (1) can be directly used with any ontology editor supporting SPIN-compliant reasoning engine to simulate ontology evolution scenario, (2) can be directly used with TopBraid composer to support form-based population of ontology templates and answer all requirements defined by **CQ 1-19**.

4.3 Prototype Implementation of the Reasoning Service

We created a prototype implementation of reasoning service over ontology templates as well as set of tools to manage templates within command-line interface [2].

It is written in Java and uses Pellet query engine [53] to evaluate SPARQL [56] and SPARQL-DL queries [2].

Ontology templates are encoded using syntax of "construct template" of SPARQL construct query [56]. Instantiation sets can be defined by a set of generating and validating SPARQL / SPARQL-DL queries . The definitions of ontology templates and instantiation sets are attached to an ontology using OWL2 compliant annotations.

There are 2 types of validating queries that we differentiate in implementation: *flat validating queries* – can validate ontology templates only with each other; *multidimensional validating queries* – can in addition formulate constraints including more groundings of one template. More formally, *flat validation query* represents instantiation set $\mathcal{S}_\mathcal{V}$ such that each $\Sigma \in \mathcal{S}_\mathcal{V}$ is set of grounding substitutions of mutually different *ontology templates* (i.e. $\forall \sigma_i, \sigma_j \in \Sigma : (\sigma_i \neq \sigma_j) \implies template(\sigma_i) \neq template(\sigma_j)$). Examples of both types of the queries are shown in the following sections.

Flat validating queries and *generating queries* are SPARQL / SPARQL-DL select queries over the *core ontology*. Result of a query, i.e. variable bindings is after execution transformed into instantiation set. To evaluate *multidimensional validating queries* we introduced *Evolution schema ontology*, an OWL 2 ontology that describes concrete evolution schema. Within the ontology, each ground substitution of an ontology template is represented by unique individual together with its variable mappings. Thus query over evolution scheme ontology can answer questions such as "Which *ontology template groundings* (substitutions) has variable $\$COMP_1$ assigned to value *Component* ?". Instead of core ontology, a *multidimensional query* queries dataset consisting of *core ontology* and *evolution schema ontology*. This is implemented through "group graph pattern" [56] of SPARQL language.

4.3.1 MONDIS project use case

Ontology templates are currently used within the research project MONDIS[3]. Aim of the project is to create a knowledge-based system for description of monuments, their damage analysis, intervention planning and prevention in the field of cultural heritage protection. Recall the example ontology from Section 3.2 about components and materials and their templates T_1, T_2:

---

[2]http://kbss.felk.cvut.cz/web/portal/web/blaskmir/ontology-templates, cit. 10.8.2015
[3]http://www.mondis.cz, cit. 10.8.2015

$COMP_1$	MAT_1
Pillar	*Wood*
Pillar	*OrganicMaterial*
VerticalElement	*Wood*
VerticalElement	*OrganicMaterial*

Table 4.1: Result of generating query for T_1

The example ontology is proper fragment of Monument Damage Ontology [8, 57] developed within the project MONDIS. The ontology templates T_1 and T_2 were used by domain experts to collect general knowledge about sub-concepts of *Component* and *Material*.

The generating query used for the template T_1 was :

```
SELECT $COMP1 $MAT1
WHERE {
    $COMP1 rdf:subClassOf Component .
    $MAT1 rdf:subClassOf Material .
    FILTER (! ($COMP1 = Component) )
    FILTER (! ($MAT1 = Material) )
}
}
```

The SPARQL-DL query over the *core ontology* returns variable bindings as shown in Table 4.1. Each row of the query result can be naturally transformed to instantiation with one substitution (e.g. second row of the table is $\{\sigma\}$ where $\sigma = \{[\$COMP_1/Pillar], [\$MAT_1/Wood]\}$. Same query, but with appropriate variable names was used for generating query of T_2.

One of the validating query was query :

```
SELECT $COMP1 $MAT1 $COMP2 $MAT2
WHERE {
    $COMP2 rdf:subClassOf $COMP1 .
    $MAT2 rdf:subClassOf $MAT1 .
}
```

Each row of the validating query result can be transformed into instantiation consisting of 2 substitutions $\{\sigma_i, \sigma_j\}$, e.g. :

○ $\sigma_i = \{[\$COMP_1/Pillar], [\$MAT_1/Wood]\}$

○ $\sigma_j = \{[\$COMP_2/Pillar], [\$MAT_2/OM]\}$

The generating query of T_1, generating query of T_2 and the validating query are actually not evaluated alone as the combined instantiation can be evaluated by one SPARQL query. On the other hand, if $\mathcal{S}_1$ and $\mathcal{S}_2$ are resulting instantiation sets of the generating queries w.r.t. T_1 and T_2, the canonical $\mathcal{S}_\mathcal{G}$ can be constructed by $\mathcal{P}(\mathcal{S}_1 \cup \mathcal{S}_2)$. Let $\mathcal{S}'$ be resulting instantiation set of the validation query. The validating instantiation set $\mathcal{S}_\mathcal{V}$ can be constructed by splitting

each substitution within $\mathcal{S}'$ into two ground substitutions, i.e. $\mathcal{S}_{\mathcal{V}} = \{\{\sigma_1, \sigma_2\} \mid \sigma \in \mathcal{S}' \wedge \sigma_1 = \sigma|_{var(T_1)} \wedge \sigma_2 = \sigma|_{var(T_2)}\}$.

Within the MONDIS project, similar templates were created also between other top level concepts that are related to diagnosis and intervention of damages (i.e. concept *ManifestationOfDamage*, *Agent*, *Mechanism*, *Intervention*) [57]. In addition to this general knowledge, some real object descriptions (e.g. about "north tower of Prague's castle and its crack") were collected using concept and role assertions.

4.3.2 P3P use case

Validating queries will be demonstrated on use case from domain of privacy protection. It will be also shown how two groundings of one ontology template can generate inconsistencies. P3P [58] is W3C recommendation for expressing service privacy practices. The practices are described by XML-based P3P policies in terms of data that will be collected, purposes for which the data will be used, a period how long the data will be held etc. In the paper [59], we proposed framework for detection and explanation of P3P policy inconsistencies using semantic technologies. The policies are transformed to an ontology and then validated by ontology consistency check and set of SPARQL-DL queries. This was implemented in P3P privacy policy editor [60]. Although P3P editor was able to find inconsistencies we were not able to provide user human-readable message to explain inconsistency as we did not know about them in advance. This can be done by framework described in this thesis.

One of the ontology templates (T) used in our P3P framework is the following :

$$\$\textbf{DATA} \sqsubseteq \exists hasPurpose.(\$\textbf{PUR} \sqcap$$
$$\exists hasRequirement.\$\textbf{REQ})$$
$$\$\textbf{DATA} \sqsubseteq \forall hasPurpose.((\neg\$\textbf{PUR})$$
$$\sqcup \forall hasRequirement.\$\textbf{REQ})$$
$$\$\textbf{DATA\_TR} \sqsubseteq \exists hasPurpose.\$\textbf{PUR}$$
$$\$\textbf{DATA\_TR} \sqsubseteq \forall hasPurpose.((\neg\$\textbf{PUR})\sqcup$$
$$\forall hasRequirement.(Always\sqcup$$
$$\$\textbf{REQ}_1 \sqcup \$\textbf{REQ}))$$

The ontology template T is used to state that some data ($\$DATA$) such as "User's business contact information" is collected for some purpose ($\$PUR$) such as "Marketing of services or products". The $\$REQ$ variable indicates requirement to which extent is the purpose required for the service. The requirement can be one of 3 values : *opt-in* - the user has to affirmatively request usage of data for this purpose; *opt-out* - the data may be collected for this purpose unless the user requests otherwise; *always* - the data will be used for this purpose.

Variables $\$DATA\_TR$ and $\$REQ_1$ can be generated from variables $\$DATA$ and $\$REQ$, respectively. A validating query is expressed in SPARQL query, which can well describe the

syntactic nature of generated dependencies. Validation query can be expressed using dataset of core ontology and controlled evolution scenaro ontology as follows :

SELECT $IS1$ $IS2$ **WHERE** {

```
  # query  to  evolution  scenario  ontology
  GRAPH p3p:evolutionSchema {
   $IS1  arg:DATA_TR  $DATA_TR1 ;
         arg:PUR  $PUR ;
         arg:REQ  $REQ1 .
   $IS2  arg:DATA_TR  $DATA_TR2 ;
         arg:PUR  $PUR ;
         arg:REQ  $REQ2 .
  }

  # query  to  core  ontology
  GRAPH p3p: {
    $DATA_TR1
       (rdf:subClassOf)+ $DATA_TR2 .
  }
  FILTER(isStronger($REQ1, $REQ2))
}
```

The first part of the query is evaluated on the evolution scenario ontology. It selects all instantiations IS_1 and IS_2 such that their purposes are same value. PUR is bound to this value. $DATA\_TR_1$ binds to "DATA_TR" variable of original template. Similarly, $DATA\_TR_2$, REQ_1 and REQ_2 are bound. The second part of the query is evaluated on the core ontology, i.e. the P3P ontology. *isStronger* is SPIN-based SPARQL function that relates requirements, it is true if used for arguments ⟨always, opt-out⟩, ⟨always, opt-in⟩, ⟨opt-out, opt-in⟩ and false otherwise. Transformation of the query to the validating instantiation set is straight-forward as each row of the query result is translated to one instantiation $\{\sigma_1, \sigma_2\}$ directly corresponding to values of IS_1 and IS_2.

4.4 Experimental Evaluation of the Reasoning Service

Section 3.2 introduced two algorithms to compute consistency of evolution schema i.e. Algorithm 3 and Algorithm 5. The first algorithm computes consistency of the schema in straightforward way. It checks correctness of all validation constraints and then compute maximal instantiations that should be possible to add to core ontology without becoming inconsistent. As stated in Section 3.3.2, this is well known NP-complete problem of finding all minimal hitting sets of a set. The hitting set is computed from union of all instantiations created from generating and validating constraints. On the other hand, the Algorithm 5 computes it other way around. It precomputes MIPS from groundings of all possible instantiations and then checks that they

are correctly expressed by generating and validating constraints. In the following text we will compare those algorithms on provided P3P and MONDIS examples.

4.4.1 Initial Setting of The Experiment

From both examples we provided 5 different size core ontologies. Such ontologies were queried as described in previous chapter in order to create instantiations and related groundings. The queries were executed on the core ontologies, thus the core ontologies represent as well the scenario context of the evolution. Each of the examples has one validation constraint that was executed in order to retrieve validating instantiations as described in the previous text. The Algorithm 3 (referred as "main algorithm") for computation of consistency schema uses hitting set algorithm. We will differentiate following hitting set algorithms used for the evaluation (see Section 2.1 for overview of the algorithms):

- **reiter** – the Reiter's algorithm [14].

- **bool-lin-jiang** – the Boolean Algebra algorithm published by by Lin and Jang [43].

- **boolean-algebra** – the Boolean Algebra algorithm optimized by Pill et al. [44].

- **BHS-tree** – the BHS-tree algorithm [43] that is based on a binary tree.

- **staccato** – the Staccato algorithm [45] that is based on a binary matrix.

The Algorithm 5 (referred as "alternative algorithm") uses Pellet reasoner to compute diagnoses on "modification closure ontology". In the evaluation we will compare only evaluations of Pellet's computation of all diagnoses with computation of the mentioned hitting set algorithms. Other computation times are irrelevant as every consistency check used within the evaluation took at most 200 ms. For the complex cases, this is with respect to overall computation time irrelevant. By label **pellet** we will represent computation time to generate all diagnoses of size **pellet-diagnoses**. If the hitting set algorithm or Pellet's computation of diagnoses does not finish in provided time we will represent it as "–". In case of Pellet due to incremental generation of explanations we can output set of computed diagnoses even if the algorithm does not terminate.

We will use following variables that represents the structures of inputs and outputs of the evaluations:

- **core-classes** – represents number of OWL2 classes OWL2 within the core ontology.

- **core-axioms** – represents number of OWL2 axioms within the core ontology.

- **closure-axioms** – represents number of OWL2 axioms within the "modification closure ontology", i.e. the ontology that was created by adding all possible groundings of ontology templates.

- **otis** – represents number of instantiations whose groundings were added to core ontology in order to evaluate the scenario.

	core-classes	core-axioms	closure-axioms	otis	oti-conflicts	oti-diagnoses
O1	7	12	66	12	12	18
O2	8	14	86	16	15	54
O3	11	20	146	28	36	450
O4	12	22	211	42	48	18900
O5	13	24	240	64	52	

Table 4.2: Input and output structures used in evaluation for MONIS example ontology.

	reiter	bool-lin-jiang	BHS-tree	boolean-algebra	staccato	pellet	pellet-diagnoses
O1	15	10	11	9	255	95	7
O2	54	46	19	13	1141	127	9
O3	1107	165	112	69	–	253	16
O4	–	174200	243339	38946	–	302	23
O5	–	–	–	–	–	450	25

Table 4.3: Computation times of the MONDIS evaluation.

- ○ **oti-conflicts** – represents number of instantiations of validation constraints, i.e. number of conflicts that validation constraints describes over "modification closure ontology" .

- ○ **oti-diagnoses** – represents number of minimal diagnoses that were generated based on validation constrains using the hitting set algorithm.

Each evaluation on the ontology was computed 5 times and the average value was computed. For the hitting set algorithms as well as for the Pellet's computations we set a hard limit of 20 minutes. If the computation does not finish by this time, it is terminated. All the hitting set algorithms were implemented in JAVA based on descriptions provided by the sreferences. It was evaluated on SGI 2200 Origin, CPU 4400 Mhz, MIPS R12000 (IP27) processors, main memory 1 GB.

4.4.2 MONDIS Example

We evaluated consistency schema algorithms on 5 examples of ontologies based on MONDIS example templates used within this thesis, i.e. :

- ○ $T_1 = \{\$COMP_1 \sqsubseteq \forall hasMaterial.\neg\$MAT_1 \}$

- ○ $T_2 = \{\_c : \$COMP_2 \sqcap \exists hasMaterial.\$MAT_2 \}$

The core ontologies contained only taxonomy of concepts (i.e. only subClassOf axioms between atomic OWL2 classes). We used one validation constraint, i.e. the SPARQL query described in Section . Description on input and output structures of the evaluation is provided in Table 4.2, while the computation times in milliseconds are provided in Table 4.3.

	core-classes	core-axioms	closure-axioms	otis	oti-conflicts	oti-diagnoses
O1	36	83	183	6	9	6
O2	37	85	250	12	18	36
O3	39	90	321	18	30	196
O4	41	95	392	24	48	625
O5	44	102	626	45	99	

Table 4.4: Input and output structures used in evaluation for P3P example ontology.

4.4.3 P3P Example

We evaluated consistency schema algorithms on 5 examples of ontologies based on P3P example templates used within this thesis, i.e. :

One of the ontology templates (T) used in our P3P framework is the following :

$$\$\textbf{DATA} \sqsubseteq \exists hasPurpose.(\$\textbf{PUR} \sqcap$$
$$\exists hasRequirement.\$\textbf{REQ})$$
$$\$\textbf{DATA} \sqsubseteq \forall hasPurpose.((\neg\$\textbf{PUR})$$
$$\sqcup \forall hasRequirement.\$\textbf{REQ})$$
$$\$\textbf{DATA\_TR} \sqsubseteq \exists hasPurpose.\$\textbf{PUR}$$
$$\$\textbf{DATA\_TR} \sqsubseteq \forall hasPurpose.((\neg\$\textbf{PUR})\sqcup$$
$$\forall hasRequirement.(Always \sqcup$$
$$\$\textbf{REQ}_1 \sqcup \$\textbf{REQ}))$$

The smallest P3P ontology i.e. O1 contained 83 axioms, from which (a) 39 logical axioms, (b) 36 OWL2 classes, (c) 7 OWL2 object properties, (d) 30 subClassOf axioms, (e) 6 equivalent-Classes axioms, (f) 2 DisjointClasses axioms, (g) 6 general concept inclusion axioms and (g) 1 object property range axiom. Thus the ontology has expresivity of $\mathcal{ALC}$ $\mathcal{DL}$.

In the case of P3P we needed to check not only consistency of ontology but also satisfiability of classes. This can be simulated by adding one individual per each class. Such ontology is consistent if and only if original ontology is consistent and has all classes satisfiable. Description on input and output structures of the evaluation is provided in Table 4.4, while the computation times in milliseconds are provided in Table 4.5.

4.4.4 Results

The results of the evaluation shows that the "main approach" of computing schema consistency is useful in cases where core ontology is of big size. This is due to the fact that computation of the hitting sets are not dependent on the core ontology. The "alternative approach" is most-likely good choice in cases where the core ontology is small. It is shown in the MONDIS example that it outperforms the "main approach". In the P3P case, the "alternative approach" did not terminate

	reiter	bool-lin-jiang	BHS-tree	boolean-algebra	staccato	pellet	pellet-diagnoses
O1	9	11	12	9	21	–	67
O2	24	11	17	13	1986	–	103
O3	95	28	46	34	–	–	101
O4	3165	91	300	113	–	–	98
O5		–	–	–	–	–	87

Table 4.5: Computation times of the P3P evaluation

even with the smallest ontology. Even though it is useful as it provides a lot of justifications of the inconsistencies which may detect inconsistency in the evolution schema. An interesting idea would be to use algorithms for computing MIPS, that depends on the order of input axioms such as CS-trees [61]. If such axioms are ordered "correctly" they generate diagnoses faster. The knowledge about relation between ontology templates and their groundings could help order the input set of axioms for such algorithms.

5 Use Cases

This chapter presents two cases where ontology templates were used. First section describe use of the templates in MONDIS project while second section describe full design of system based on ontology templates.

5.1 MONDIS Use Case

In the MONDIS project, the ontology developed in a close cooperation of a domain expert with a knowledge engineer needs to be enriched mainly with the knowledge of new protection techniques. However, form the manageability point of view, it is necessary to get rid off the presence of the knowledge engineer in the evolution process. The only applicable approach is to prepare evolution scenarios expressed in terms of ontology templates that allow the domain engineer himself to evolve the ontology in a consistent way. Realization of scenarios were done in Ontomind – a mindmapping software developed within the project [54]. Creation of the mindmaps within the tool is associated with generation of ontological axioms. SPARQL-based ontology of controlled evolution was used here to guide user within the process of creation, explaining inconsistencies immediately when they occurs.

5.2 P3P use case

This section presents design of service for validating of P3P policies that uses ontology templates in translation process. At first we present a mechanism for translating a P3P policy into a policy ontology (i.e. core ontology in context controlled evolution scenario) and a set of integrity constraints (i.e. generating constraints in context of controlled evolution scenario) in the form of SPARQL-DL queries. If a policy ontology created by the translation is consistent, all of its classes satisfiable and all integrity constraint satisied, the original P3P policy is "semantically consistent", i.e. it can be used by the service. If any of these tests fails, the P3P policy is "semantically inconsistent", i.e. it is not possible for the service to be compliant with the policy. Translation from a P3P policy to a policy ontology can be divided into three steps. First, we load a generic policy ontology - a common upper ontology of all P3P policies. Second, we translate each statement of the P3P policy into a set of OWL axioms that pose restrictions to some classes of the generic policy ontology. Third, we check all integrity constrains. In the following three sections we describe the generic policy ontology. Later in the text, we describe a mechanism

for translating P3P policy statements into policy ontology axioms and a mechanism for creating integrity constraints using SPARQL-DL queries. Last section shown support for editing

5.2.1 P3P Policy ontology design

In this section we present the mechanism for translating a P3P policy into a policy ontology and a set of integrity constraints in the form of SPARQL-DL queries. If a policy ontology created by the translation is consistent, all of its classes satisfiable and all integrity constraints satisfied, the the original P3P policy is "semantically consistent", i.e. it can be fulfilled by the service. If any of these tests fail, the P3P policy is "semantically inconsistent", i.e. it is not possible for the service to be compliant with the policy. Translation from a P3P policy to a policy ontology can be divided into three steps. First, we load a generic policy ontology - a common upper ontology of all P3P policies. Second, we translate each statement of the P3P policy into a set of OWL axioms that pose restrictions to some classes of the generic policy ontology. Third, we check all integrity constrains. In the following three sections we describe the generic policy ontology. Later, in the text, we describe a mechanism for translating P3P policy statements into policy ontology axioms and a mechanism for creating integrity constraints using SPARQL-DL queries.

5.2.1.1 Basic structure of the policy ontology

To keep the reasoning as efficient as possible (in this case to preserve the pseudo-tree model property of the designed ontology), we decided to use only class and property taxonomies for axiomatization. Basic structure of the P3P generic policy ontology is shown in Figure 5.5:

- *Data* - represents a set of all P3P datatypes. Each P3P datatype corresponds to a subclass of the *Data* class. (e.g. *User.home-info* is a subclass of the *Data* class).

- *Category* - represents a set of all P3P categories. Each P3P category corresponds to a subclass of the *Category* class. (e.g. *Physical* is subclass of the *Category* class)

- *Purpose, Recipient* and *Retention* - describes a set of all its P3P purposes (recipients, retentions). Each of the P3P purposes (recipients, retentions) correspond to a subclass of *Purpose* (*Recipient, Retention*) class. All subclasses of the *Purpose* (*Recipient, Retention*) class are disjoint classes. (e.g. both *Telemarketing* and *Contact* are subclasses of the *Purpose* class and *Telemarketing* is disjoint with *Contact*, etc.)

- *CollectionType* - corresponds to the optional attribute of the P3P datatype element. Its two direct subclasses *Optional* and *Mandatory* specify whether the collection of a P3P datatype is optional or mandatory.

- *Requirement* - corresponds to the required attribute of the P3P purpose or the P3P recipient. Its direct subclasses specify to which extent the purpose/recipient is required for the service. It has 3 mutually disjoint subclasses: *Opt-in*, *Opt-out* and *Always* which correspond to P3P values of the required attribute.
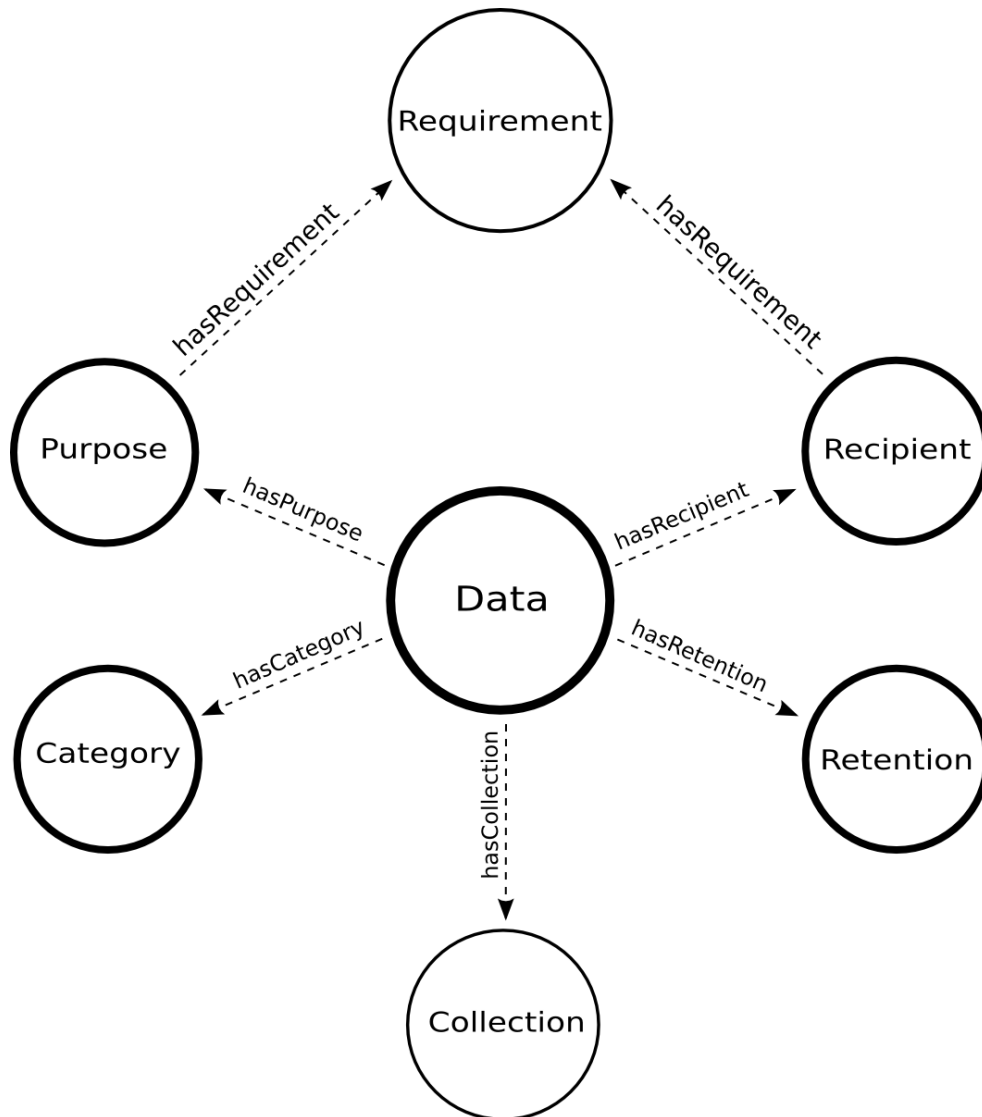
50

Figure 5.1: Basic structure of the P3P policy ontology. Circles represents basic classes, arrows represents object properties pointing from its domain to its range, e.g. The domain of the object property *hasRecipient* is *Data* class, while its range is *Recipient* class.

```
BasicData
  User_RP
    User
    User.home-info_RP
      User.home-info
      User.home-info.postal_RP
        User.home-info.postal
        ...
      User.home-info.online_RP
        User.home-info.online
        ...
```

Figure 5.2: Taxonomy defined by datatype classes

The figure additionally illustrates how the top-level classes are connected to each other by object properties. The central class of the figure is the *Data* class. The *Data* class references the *Category*, *Purpose*, *Recipient*, *Retention* and *CollectionType* classes by object properties *hasPurpose*, *hasRecipient*, *hasRetention*, *hasCategory* and *hasCollectionType* respectively. Note the direct correspondence of described object properties with database schema relations (d-purpose, d-recipient, etc.) described in Section 2.1. Additionally, both the *Purpose* and the *Recipient* class reference the *Requirement* class by the object property *hasRequirement*.

5.2.2 Datatype taxonomies

P3P defines a hierarchy of datatypes that specify what information is collected from a user. For example, a service can state to collect datatypes `#user.home-info.postal` and `#user.home-info.online` which would mean that any information about the postal and online address of a user may be collected. Similarly, a service can state to collect the datatype `#user.home-info`, which would mean that any information about the user's home contact information including `#user.home-info.postal`, `#user.home-info.online`, and `#user.home-info.telecom` (telecomunication information) may be collected.

For modeling datatypes, the part-whole relation is an appropriate choice. For each datatype x that is an immediate part of a datatype y, an axiom x isPartOf some y can be defined. Then for each datatype, its reflexive part class is created (we will label it by *_RP* suffix), i.e. the class representing the given datatype and all datatypes that are its parts (transitively). Thus, one could add axiom *x_RP* EquivalentTo (*x* or *isPartOf* some *x*) for each datatype *x* and define *isPartOf* as a transitive object property by axiom trans(*isPartOf*). Such a set of axioms would lead, after classification, to the taxonomy shown in Figure 5.2).

Taxonomy in the figure is particularly useful for our modeling purposes. For each P3P datatype "#x" we have two representative classes in the policy ontology. The first, own datatype class (x) that can be used whenever we need to restrict a particular datatype class without affecting its datatype parts. On the contrary, the second, reflexive part of datatype (x_RP) can be

used whenever we need to propagate the restriction applied to a given datatype to all its datatype parts. Due to performance issues we did not use the partOf object property. Instead, we created the taxonomy from Figure 5.2), explicitly by subclass axioms.

5.2.2.1 Data category propagation

In P3P, there are two different ways to address data that are collected from a user. First, a service may list particular datatypes that will be collected (e.g. `#user.home-info.postal`). Second, a service may list categories of data that will be collected (e.g. physical category). P3P specification defines what datatypes may be collected for each category (e.g. for the physical category it is `#user.home-info.postal.street`, `#user.home-info.telecom.telephone`, etc.).

To determine what datatypes belong to which category we use the *hasCategory* object property and state *x* subClassOf *hasCategory* some *y*, for each datatype *x* that is in the category *y*. Additionally, for further evaluation of integrity constraints we need to be able to find out what categories might be collected if a service collects a particular datatype.

If we consider the example where a service collects only `#user.home-info.postal` datatype, since `#user.home-info.postal.street` is in data category physical and `#user.home-info.postal.city` is in category demographic, a service may collect both physical and demographic data categories. For this purpose we introduce a new transitive object property *inheritsCategoriesFrom*. Both the domain and the range of the property is *Data* class. The object property is very similar to the *hasPart* object property (i.e. the inverse of *isPartOf* object property). The following axioms demonstrate how the object property *inheritsCategoriesFrom* is used for the datatype classes from the above example:

- ○ *User.home-info.postal.street* subClassOf *hasCategory* some *Physical*

- ○ *User.home-info.postal.city* subClassOf *hasCategory* some *Demographic*

- ○ *User.home-info.postal* subClassOf
       inheritsCategoriesFrom some *User.home-info.postal.street*

- ○ *User.home-info.postal* subClassOf
       inheritsCategoriesFrom some *User.home-info.postal.city*

- ○ transitive(*inheritsCategoriesFrom*)

The policy additionaly contains classes *DirectPhysicalData* and *InferredPhysicalData* defined as follows :

- ○ *DirectPhysicalData* equivalentTo *hasCategory* some *Physical*

- ○ *InferredPhysicalData* equivalentTo *DirectPhysicalData* or (*inheritsCategoriesFrom* some *DirectPhysicalData*)

53

```
CategorizedData
  InferredDemographicData
    User
    User.home-info
    User.home-info.postal
    DirectDemographicData
      User.home-info.postal.city
  InferredPhysicalData
    User
    User.home-info
    User.home-info.postal
    DirectPhysicalData
      User.home-info.postal.street
  ...
```

Figure 5.3: Taxonomy of direct and inherited categories

The class *DirectPhysicalData* is a superclass of all datatype class es for which category *Physical* was asserted from the P3P specification. The class *InferredPhysicalData* is a superclass of all datatype classes whose parts might collect some datatypes that are from *Physical* categories. To see the classes after classification see Figure 5.3.

5.2.2.2 P3P policy translation algorithm

Previous sections of this chapter described the generic policy ontology that defines the basic structure of P3P, datatype taxonomies and category propagation. This ontology is imported into a new ontology that represents a particular P3P policy.

The algorithm for translation a P3P policy into a policy ontology is shown in Figure 5.4. It accepts the generic policy ontology and a P3P policy as inputs and outputs a new policy ontology that corresponds to the original P3P policy. At first, the algorithm imports the generic policy ontology into the newly created ontology (line 1-2). Then, a new ontology axiom set is created from each P3P statement as follows. According to data-centric semantics explained in Section 2.1, every P3P policy can be translated into five database schema relations. To simplify the translation principle, assume that we add a special axiom to our axiom set for every instance of each relation occurring in the P3P statement.

We can demonstrate it with an example from Figure 2.1. It is stated in the first statement that `#user.home-info` is collected for the purpose contact that may be opt-out and for the purpose individual-analysis that is always required. The recipient of the data is ours and is always required. The P3P statement translation generates the following axioms:

- *User.home-info* subClassOf *hasPurpose* some
  (*Contact* and *hasRequirement* some *Opt-out*)

1: **function** TRANSLATE(Gen_O, Pol)
2: **Input:** Gen_O generic policy ontology, *Pol* P3P policy.
3: **Output:** semantically valid policy ontology PO corresponding to the P3P policy Pol, or an error whenever an inconsistency is found.
4:      create an empty policy ontology P_O
5:      import Gen_O to P_O
6:      for each P3P statement $S \in Pol$ do
7:           create an ontology axiom set *AS* from S
8:           check *AS* for statement constraints
9:           add each axiom from *AS* to ontology P_O
10:          check P_O for policy inconsistencies
11:     **return** P_O
12: **end function**

Figure 5.4: Algorithm for checking the consistency of P3P policy. The algorithm creates and imports the generic policy ontology into the newly created ontology (lines 4-5). Each P3P statement is translated to the axiom set, which is checked for statement-scope inconsistencies (lines 7-8), then it is added to the new ontology and policy-scope inconsistencies are checked (lines 9-10).

  ○ *User.home-info* subClassOf *hasPurpose* some
        (*Individual-analysis* and *hasRequirement* some *Always*)

  ○ *User.home-info* subClassOf *hasRecipient* some
        (*Ours* and *hasRequirement* some *Always*)

These axioms are basic axioms of the translation procedure. In order to express policy model restrictions discussed in Section 2.1, more axioms need to be added which is discussed in the following sections. Continuing the explanation of the algorithm from Figure 5.4, the next line (line 5) checks statement- scope restrictions discussed in section Section 2.1. If P3P statement-scope restrictions are violated, the algorithm stops and points to the particular condition that was not satisfied. Otherwise, the algorithm continues and incrementally adds axioms to the final policy ontology (line 6), while continually checking the consistency of the ontology. By checking the consistency of the ontology, some of the policy-scope restrictions (see Section 2.1) are checked. If the ontology becomes inconsistent or some of the classes become unsatisfiable, the algorithm terminates and the last axiom that was added is identified causing the problem. Otherwise, the algorithm continues and processes other policy-scope restrictions (line 7) that are expressed by means of SPARQL-DL queries. If all the query checks pass it is guaranteed that the new policy ontology is valid according to the model discussed in Section 2.1 and it is returned by the algorithm.

5.2.2.3 Policy model restrictions expressed in OWL

Ontology axioms in the policy ontology are used for expressing both global-scope and policy-scope restrictions that were discussed in Section 2.1. All global-scope restrictions were expressed in the generic policy ontology. An example of such a restriction is the propagation of P3P categories through the object property *inheritsCategoriesFrom*.

Policy-scope restrictions used in the policy ontology have one of three forms:

○ subclass axioms restricting a datatype class (e.g. class *User.home-info*),

○ subclass axioms restricting reflexive parts of a datatype class (e.g. class *User.home-info_RP*),

○ general inclusion axioms (GCI).

An example of policy-scope restrictions that use OWL axioms to restrict a datatype class are data-centric (functional) constraints (see Section 2.1). Data-centric constraints of the first statement of the policy from Figure 2.1 are translated as follows:

○ *User.home-info* subClassOf *hasPurpose* only
    ((not *Contact*) or *hasRequirement* only *Opt-out*)

○ *User.home-info* subClassOf *hasPurpose* only
    ((not *Individual-analysis*) or *hasRequirement* only *Always*)

○ *User.home-info* subClassOf *hasRecipient* only
    ((not *Our*) or *hasRequirement* only *Always*)

The first OWL axiom ensures that if any other P3P statement states that `#user.home-info` is collected for the *Contact* purpose, its requirement must be *Opt-out*. If this condition is violated, class *User.home-info* will become unsatisfiable. Similarly, the second and the third OWL axiom is created.

An example of P3P policy-scope restrictions that use OWL statements restricting reflexive part of the datatype class are all data hierarchy constraints (see Section 2.1). The first statement of the policy from Figure 2.1 expresses data hierarchy constraints as follows:

○ *User.home-info_RP* subClassOf *hasPurpose* some *Contact*

○ *User.home-info_RP* subClassOf *hasPurpose* only
    ((not *Contact*) or (*hasRequirement* only (*Opt-out* or *Always*)))

○ *User.home-info_RP* subClassOf *hasPurpose* some *Individual-analysis*

○ *User.home-info_RP* subClassOf *hasPurpose* only
    ((not *Individual-analysis*) or (*hasRequirement* only *Always*))

○ *User.home-info_RP* subClassOf *hasRecipient* some *Ours*

○ *User.home-info_RP* subClassOf *hasRecipient* only
((not *Ours*) or (*hasRequirement* only *Always*))

The first OWL axiom states that contact purpose should be propagated to all sub-parts of the `#user.home-info` datatype. The second OWL axiom states that sub-parts of the datatype can have contact purpose either opt-out or required (always). This pair of OWL axioms ensures that the data hierarchy constraint for the contact purpose holds. Similarly, the next two pairs of OWL axioms ensure that data hierarchy constraints hold for the purpose individual-analysis and the recipient ours.

5.2.2.4 Policy model restrictions expressed in SPARQL-DL

Some of the policy model restrictions are natural to the model with respect to the close-world assumption. For such cases we used the SPARQL-DL query language. The queries are used to express some statement-scope restrictions and policy-scope restrictions.

As an example of a statement-scope restriction, consider the constraint defined in Section 2.1. The constraint defines that if a collection of some datatypes in a statement is required, then at least one purpose must be specified as always required. The following SPARQL-DL query checks this constraint:

SELECT ?x WHERE {
        ?x subClassOf (*hasCollection* some *Required*) .
        NOT { ?x subClassOf (*hasPurpose* some
                (*hasRequirement* some *Always*) }
}

The query returns all datatype classes whose collection is required (first and second line) and for which there is not any purpose with the requirement "always" (third line). Thus, the results of the query are those datatypes that violate the constraint. If the query does not return any answers, then this policy-scope restriction is satisfied.

Note that the NOT operator is not part of the SPARQL-DL syntax, but can be encoded by a combination of OPTIONAL, FILTER and BOUND operators in SPARQL (see Negation as a Failure pattern in [26]).

To run a query for a P3P statement scoped property we have to use the generic policy ontology that is populated with only a basic translation of the P3P statement. Each statement of the P3P policy and each query may run in parallel. For a demonstration of SPARQL-DL queries in the scope of the whole P3P policy we will use a condition that restricts stated-purpose retention to a number of purposes allowed (Section 2.1) :

SELECT ?x WHERE {
        ?x subClassOf ((*hasRetention* some *Stated-purpose*)
                and (*hasPurpose* min 2))
}

The query returns all datatype classes for which at least two purposes are defined and its retention is defined as "stated-purpose". If the query returns an empty set, the restriction was not violated for any datatype.

5.2.3 P3P Privacy framework design

To validate the proposed policy model described in Section 5.2.1 we need a framework which can visualize our results and allows us to experiment with policy restrictions in an easy way. This chapter describes the design of such a framework. Section 5.2.3.1 identifies the requirements and proposes the framework which consists of four logicaly separated modules. Section 5.2.3.2 explains the necessary enrichment of the policy ontology that will be used in the framework. The rest of the chapter describes individual modules and reveals their implementation details.

5.2.3.1 Design of privacy framework

The privacy framework has to be able to process input a P3P file, transform it to the policy ontology, examine this ontology in order to find inconsistencies and explain these inconsistencies to a user in a human readable way. These requirements naturally divide the framework into four modules: selection module, transformation module, reasoning module and visualization module.

Each of the modules requires a configuration input from a user of framework, i.e. the selection module needs to define queries that will be answered on concrete P3P data; the transformation module requires a user to define what axioms should be created from the input data; the reasoning module must be configured by a user to generate the most relevant explanations of inconsistencies and visualisation module requires the user to define mapping from OWL explanations to human readable descriptions.

Clearly, the hardest part of configuring the modules lies in the definition of policy constrains which is carried out in the transformation module. It must be designed in a way that shows very clearly what the results of transformation are. Thus, it seems appropriate to represent a transformation module interface to the user as an ontology template, i.e. axioms with variables. Additionally, the structure of data that are substituted to ontology template must be as simple as possible, which brings us to design decisions of the selection module.

The selection module queries P3P input data and prepares its output to be consumed by the transformation module. To make output as simple as possible we propose that the result of the selection module is organized in a table structure. The table structure can be seen as rows of entries where each entry is a map of variables (column names) to its data. The same variables that occur in entries are then used in ontology template of the transformation module. Each row of data is translated exactly to one axiom of policy ontology.

The content of variables from the output of the selection module should either be a valid IRI [62] of an OWL entity defined in the generic policy ontology or a string constant. Therefore, the selection module needs a mechanism to map the IRI of a P3P element into the IRI of an OWL entity. This map can also be reused by the visualization module, so it seems appropriate to implement it by annotations to the generic policy ontology that are shared among the modules.

The reasoning module adds axioms generated from the transformation module and continuously checks the consistency of the ontology and the satisfiability of its classes. If needed, the axioms generated from the translation module can be annotated by other metadata (e.g. what were the variables of the data, what is the original template, etc.) to help improve explanations of inconsistency/unsatisfiability.

The following section explains the necessary enrichment of the policy ontology that will be used in the selection and visualization modules. Afterwards, a detailed description of each module follows.

5.2.3.2 Structure of enriched policy ontology

P3P 1.1 specification [17] refers to three XML schema files that allow the validation of P3P policy files, the P3P 1.1 base data schema document, the P3P 1.1 policy schema document and the P3P 1.0 policy schema document. In order to extend predefined elements of P3P files, there is an EXTENSION element defined in most of the XML elements of both P3P policy schema documents. P3P datatypes can be extended by creating a new file similar to the P3P 1.1 base data schema document, including it in the P3P 1.1 policy schema document and defining root datatype elements of newly added datatypes.

Like the structure of P3P XML schema files, the policy ontology defined in Section 5.2.1 is distributed into multiple ontology files :

○ p3p-language.owl - contains definitions of all the classes and object properties mentioned in Section 5.2.1 except for the classes that represent datatypes and their reflexive parts (e.g. definition of *User.home-info* or *User.home-info_RP* is not included). Examples of such entities are *Purpose*, *Telemarketing*, *hasRetention*, *DirectDemographicData*, *Inferred-PhysicalData*, etc.

○ p3p-bds.owl - imports p3p-language.owl ontology and contains only the definitions of all datatypes and their reflexive parts. As explained in Section 5.2.1, the definition of a datatype describes its related categories (e.g. *User.home-info* subClassOf *hasCategory Physical*) and its related datatypes (e.g. *User.home-info* inheritsCategoriesFrom some *User.home-info.postal*).

○ generic-policy.owl - imports p3p-language.owl and p3p-bds.owl. Moreover, the ontology contains general class axioms that represent policy-scope restrictions discussed in Section 5.2.1 (e.g. *hasPurpose* some *Historical* SubClassOf *hasRetention* only (not (*No-retention*))). This ontology represents generic policy ontology which is imported into the policy ontology. The policy ontology only contains axioms that are specific to the translation of concrete input P3P file.

In comparison to extension mechanisms of P3P XML schema files, extensions to generic policy ontology are more straightforward. Both, the P3P language extension and the P3P datatypes extension can be realized by the import of a new ontology into generic policy ontology.

In addition to policy ontology modules we define two ontology files that provide us P3P metadata to defined classes of the policy ontology:
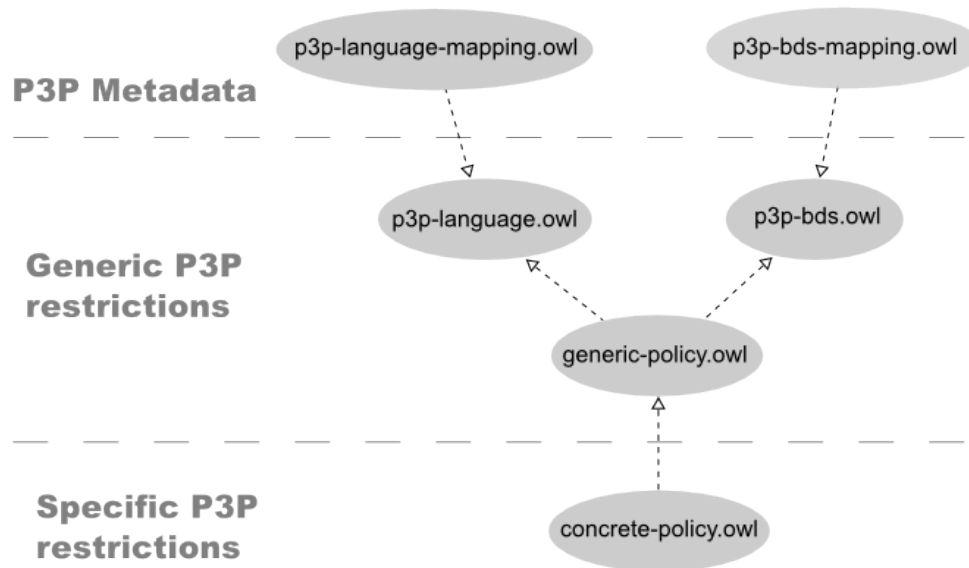
Figure 5.5: Import structure of enriched P3P policy ontology. Ovals represent ontologies, arrows represent import statements pointing from the importing ontology to the ontology being imported, e.g. the ontology from the file generic-policy.owl imports the ontology from the file p3p-language.owl.

- p3p-language-mapping.owl - imports p3p-language.owl ontology and annotates its classes with links to the corresponding definitions of P3P elements, its human readable description etc. Annotation property *hasXMLMapping* is used to link classes of the policy ontology to the elements of P3P files. As an example consider annotation *Purpose* Annotations(*hasXMLMapping*
"http://www.w3.org/2002/01/P3Pv1#PURPOSE").

- p3p-bds-mapping.owl - imports p3p-bds.owl ontology and annotates its classes with links to the corresponding elements of P3P datatype definitions, its human readable translations etc. An example of such an annotation is *User.home-info* Annotations (*hasXMLMapping* "http://www.w3.org/2006/01/P3Pv11BDS#user.home-info", *hasXMLLabel* "User's Home Contact Information"@en).

The annotation enrichment of the policy ontology not only provides helpful information while editing the ontology, but is also used in other modules of the framework. The overall structure of the policy ontology with P3P metadata annotations is described in Figure 5.5.

5.2.3.3 Selection Module

The selection module queries an input P3P file and produces variable to data mappings that are consumed by the transformation module. The input file is loaded to the selection module once, but it can be queried multiple times. Each query is specified by a selection term, that is

$$\begin{aligned}
\$ID <&- \quad id(p3p10{:}STATEMENT) \\
\$DATA <&- \quad \textit{p3p-language:BasicData} \\
\$PURPOSE <&- \quad \textit{p3p-language:Purpose} \\
\$REQUIREMENT <&- \quad \textit{p3p-language:Requirement}
\end{aligned}$$

Figure 5.6: An example of a selection term that queries for variables ID, DATA, PURPOSE and REQUIREMENT.

ID	DATA	PURPOSE	REQUIREMENT
1	*User.home-info*	*Contact*	*Opt-out*
1	*User.home-info*	*Individual-analysis*	*Always*
2	*User.home-info.online.email*	*Contact*	*Always*
2	*User.name*	*Contact*	*Always*

Figure 5.7: An example result of the selection module's query for variables ID, DATA, PURPOSE and REQUIREMENT.

a string in a meta language that the selection module understands. Ideally, the selection term should only declaratively describe the output variables of the query and its meaning in a result set. Additionally, the selection module must process P3P metadata defined in ontologies p3p-language-mapping.owl and p3p-bds-mapping.owl to be able to output the results in terms (IRIs) of the general policy ontology.

As an example of the selection term consider Figure 5.6. It contains 4 lines, where each line contains a definition of a variable and it's type (e.g. the first line defines the variable DATA with type *BasicData*). Consider the example of policy from Figure 2.1. After executing the selection term on this policy, the result set could be as described in Figure 5.7.

5.2.3.4 Transformation Module

The transformation module consumes input from the selection module and produces OWL axioms that are then loaded into the reasoning module. The module manages the set of ontology templates, where each of them is bind to some selection term. More than one ontology template can be bound to one selection term. To translate an ontology template to a set of OWL axioms - first, the selection term associated with the ontology template is executed in the selection module; second, the result is substituted to variables of ontology template.

To demonstrate, how the substitution works, consider the ontology template:

- ○ $DATA subClassOf *hasPurpose* some
  ($PURPOSE and (*hasRequirement* some $REQUIREMENT))

After substitution of data from Figure 5.7, the transformation module would produce the following axioms :

- *User.home-info* subClassOf *hasPurpose* some
  (*Contact* and (*hasRequirement* some *Opt-out*))

- *User.home-info* subClassOf *hasPurpose* some
  (*Individual-analysis* and (*hasRequirement* some *Always*))

- *User.home-info.online.email* subClassOf *hasPurpose* some
  (*Contact* and (*hasRequirement* some *Always*))

- *User.name* subClassOf *hasPurpose* some
  (*Contact* and (*hasRequirement* some *Always*))

5.2.3.5 Reasoning Module

The reasoning module adds OWL axioms generated by the transformation module to the policy ontology while testing the consistency of the ontology, the satisfiability of its classes and integrity constraints expressed in sparql-dl queries. To demonstrate implementation difficulties of the user-friendly reasoning module, consider a modified version of an example policy ontology form the Figure 2.1, where the required attribute of the contact purpose is modified to be opt-in (instead of always required). As explained in Section 2.1.2, such a policy violates data-hierarchy constrains. After the addition of all translations of ontology templates from the transformation module, the policy ontology will contain the following unsatisfiable classes: *User*, *User.home-info*, *User.home-info.online* and *User.home-info.online.email*.

The reasoning module can pick one of the datatype classes and provide an explanation of the unsatisfiability. In this case, the best choice is the most specific datatype class i.e. *User.home-info.online.email*. It provides the shortest and the most eligible explanation of the problem by following axioms:

- DisjointClasses : *Always, Opt-in, Opt-out*

- *User.home-info.online.email* subClassOf *hasPurpose* some
  (*Contact* and (*hasRequirement* some *Opt-in*))

- *User.home-info.online.email* subClassOf
  User.home-info.online.email_RP

- *User.home-info.online.email_RP* subClassOf
  User.home-info.online_RP

- *User.home-info.online_RP* subClassOf *User.home-info_RP*

- *User.home-info_RP* subClassOf *hasPurpose* only
  ((not *Contact*) or (*hasRequirement* only (*Always* or *Opt-out*)))

Even this set of axioms is too long to identify a problem. It becomes much clearer if we remove all the axioms that are defined in generic policy ontology (notice that sometimes this might not be s feasible solution). We end up with two axioms:

- *User.home-info.online.email* subClassOf *hasPurpose* some
  (*Contact* and (*hasRequirement* some *Opt-in*))

- *User.home-info_RP* subClassOf *hasPurpose* only
  ((not *Contact*) or (*hasRequirement* only (*Always* or *Opt-out*)))

The first axiom is an instance of basic axiom translation procedure. It simply states that user's email is collected for contact purpose that can be opt-in. The second axiom is an instance of data hierarchy constraints described in Section 5.2.1. The statement expresses that any of the user's home contact information can be collected and is either required to collect or its collection can be opt-out. The user's email belongs to the user's home contact information but its collection is opt-in, which is in contradiction to the previous statement.

5.2.3.6 Visualization Module

The transformation module can annotate axioms it produces in order that other modules get useful information. Such useful information would be a list of original template variables and their values. For the case discussed in the previous section we could than easily generate a human readable explanation as: "Statement 1 collects users' home contact information for contact purpose that can be opt-out. It coincides with statement 2 that collect users' email address for contact purpose that is opt-in."

5.2.4 Evolution Scenario in Ontology Editor

Consider the P3P example from the previous chapter. The validation query was fired on the following constraint violation : "User's business contact information" is collected for "Marketing of services or products" with requirement "always". "User's business telephone number" is collected for "Marketing of services or products" with requirement "opt-out".

We use the evolution scenario ontology within TopBraid Composer [28] to edit concrete instantiations of templates, i.e. instead of creating ontology templates directly in P3P ontology we create only instantiations within the evolution scenario ontology, which are then synchronized with the core ontology. It allows us to have specialized forms for each template as shown in Figure 5.8. In addition using the query above we can create SPIN constraint on instantiations that provides human-readable explanations of inconsistencies and fixes as it is shown in Figure 5.8 and Figure 5.9. The provided framework can be used with Free Edition of TopBraid Composer.
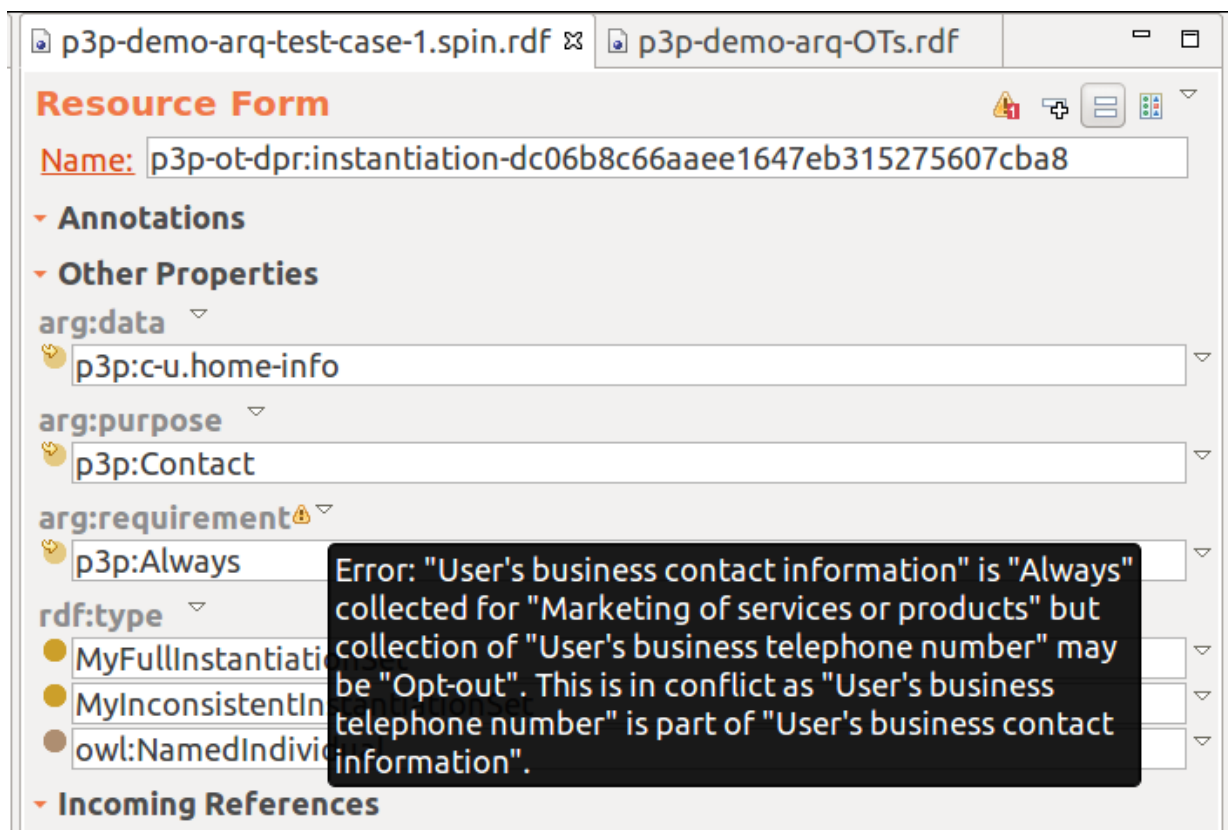
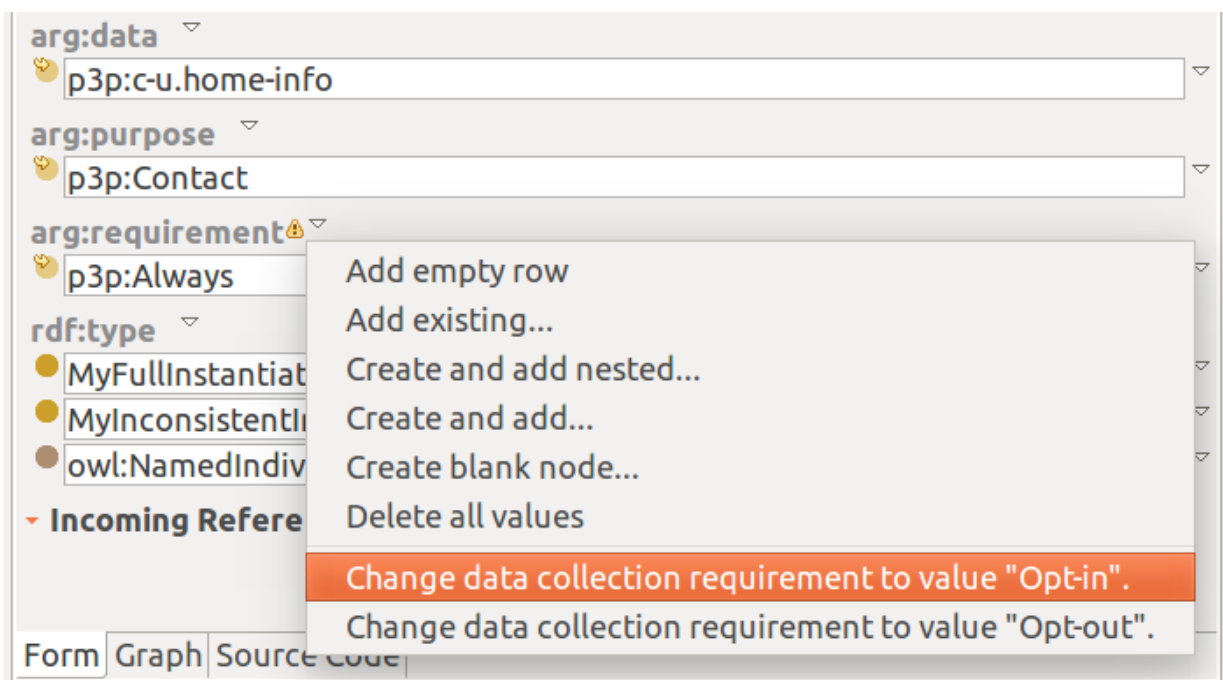Figure 5.8: P3P error in TopBraid composer.

Figure 5.9: Suggestion for fix in TopBraid composer.

6 Conclusions

This thesis has introduced a formal technique for controlling ontology evolution by domain experts or ontology engineers. The technique guarantees them that – as long as the changes they make comply with the evolution templates – the evolution preserves ontology consistency.

The formal framework only depends on the ontology consistency checking algorithm and a single MIPS algorithm for the underlying $\mathcal{DL}$. Thus, although introduced in this thesis for ALC, it can be easily extended to more expressive description logics, such as $SROIQ(D)$ [63] which is used in the current ontology standard OWL 2. Moreover it seems straight-forward to extend the framework with individual variables, role name variables, and general concept variables.

The introduced technique has a far-reaching impact on the process of ontology design and evolution. As our experience in the MONDIS project shows, domain experts feel more comfortable and less frustrated when they can predict the impact of changes they make. Another impact is the optimization of ontology engineering resources during the evolution phase.

The introduced research has opened several interesting research topics. First, we introduced two cases of ontology evolution. In the future we would like to experiment with various ontology templates w.r.t. their usability for domain experts. Compared to ontology design patterns that are meant for ontology engineers (in the same sense as software design patterns are meant for software engineers), ontology templates are also meant for domain experts – thus, they have to be simple, intuitive and have the formal guarantees introduced in this thesis.

Second, it was shown that our framework can be used not only for providing consistency within the evolution scenario, but also for specific entailments of the ontology. In the P3P case, the framework was used to check satisfiability of all classes within the scenario. It would be interesting to provide an ontology engineer with a service that would enhance these possibilities to check arbitrary entailments within the evolution, even formulated on the same level as the ontology templates are. This would provide a "query language" for an ontology engineer to check his/her assumptions of the evolving ontology and thus validate his/her intentions.

Third, our evaluation of prototype service brought up new ideas for optimization of the provided algorithms.

Bibliography

[1] Baader, F. *The description logic handbook: theory, implementation, and applications*. Cambridge university press, 2003.

[2] Sirin, E.; Parsia, B. SPARQL-DL: SPARQL Query for OWL-DL. In *OWLED*, volume 258, 2007.

[3] Kalyanpur, A. *Debugging and Repair of Owl Ontologies*. Doctoral thesis, University of Maryland at College Park, College Park, MD, USA, 2006, aAI3222483.

[4] Fernandez-Lopez, M.; Gomez-Perez, A.; Juristo, N. METHONTOLOGY: from Ontological Art towards Ontological Engineering. In *Proceedings of the AAAI97 Spring Symposium*, Stanford, USA, March 1997, pp. 33–40.

[5] Gangemi, A.; Gomez-Perez, A.; Presutti, V.; et al. Towards a Catalog of OWL-based Ontology Design Patterns. In *12th Conference of the Spanish Association for Artificial Intelligence, CAEPIA 2007*, Salamanca (Spain): AEPIA, 2007. Available from: `http://www.neon-project.org/web-content/images/Publications/caepia-catalogpatterns-vfinal.pdf`

[6] Cacciotti, R.; Valach, J.; Kuneš, P.; et al. Knowledge-based system for documentation and mitigation of damages in historical structures. *CESB13-Central Europe towards sustainable building*, 2013.

[7] Cacciotti, R.; Valach, J.; Kuneš, P.; et al. Monument damage Information System (MONDIS), An Ontological Approach to Cultural Heritage Documentation. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, volume 2, no. 5, 2013: pp. 55–60.

[8] Blaško, M.; Cacciotti, R.; Křemen, P.; et al. Monument damage ontology. In *Progress in Cultural Heritage Preservation*, Springer, 2012, pp. 221–230.

[9] Gruber, T. R. A Translation Approach to Portable Ontology Specifications. *Knowl. Acquis.*, volume 5, no. 2, June 1993: pp. 199–220, ISSN 1042-8143, doi:10.1006/knac.1993.1008. Available from: `http://dx.doi.org/10.1006/knac.1993.1008`

[10] Patel-Schneider, P.; Parsia, B.; Motik, B. OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax (Second Edition). W3C recommendation, W3C, Dec. 2012, http://www.w3.org/TR/2012/REC-owl2-syntax-20121211/.

[11] Schmidt-Schauß, M.; Smolka, G. Attributive Concept Descriptions with Complements. *Artif. Intell.*, volume 48, no. 1, 1991: pp. 1–26, doi:10.1016/0004-3702(91)90078-X. Available from: `http://dx.doi.org/10.1016/0004-3702(91)90078-X`

[12] Kalyanpur, A.; Parsia, B.; Sirin, E.; et al. Repairing Unsatisfiable Concepts in OWL Ontologies. In *ESWC, Lecture Notes in Computer Science*, volume 4011, edited by Y. Sure; J. Domingue, Springer, 2006, ISBN 3-540-34544-2, pp. 170–184.

[13] Kalyanpur, A.; Parsia, B.; Sirin, E.; et al. Debugging Unsatisfiable Classes in OWL Ontologies. *Web Semantics: Science, Services and Agents on the World Wide Web*, volume 3, no. 4, 2005, ISSN 1570-8268. Available from: `http://www.websemanticsjournal.org/index.php/ps/article/view/77`

[14] Reiter, R. A theory of diagnosis from first principles. *Artificial intelligence*, volume 32, no. 1, 1987: pp. 57–95.

[15] Karp, R. Reducibility among Combinatorial Problems. In *Complexity of Computer Computations*, edited by R. Miller; J. Thatcher; J. Bohlinger, The IBM Research Symposia Series, Springer US, 1972, ISBN 978-1-4684-2003-6, pp. 85–103, doi:10.1007/978-1-4684-2001-2_9. Available from: `http://dx.doi.org/10.1007/978-1-4684-2001-2_9`

[16] Marchiori, M. The Platform for Privacy Preferences 1.0 (P3P1.0) Specification. W3C recommendation, W3C, Apr. 2002, http://www.w3.org/TR/2002/REC-P3P-20020416/.

[17] Wenning, R.; Schunter, M. The Platform for Privacy Preferences 1.1 (P3P1.1) Specification. W3C note, W3C, Nov. 2006, http://www.w3.org/TR/2006/NOTE-P3P11-20061113/.

[18] Li, N.; Yu, T.; Anton, A. A semantics based approach to privacy languages. *Computer Systems Science and Engineering*, volume 21, no. 5, 2006: p. 339.

[19] Yu, T.; Li, N.; Antón, A. I. A formal semantics for P3P. In *Proceedings of the 2004 workshop on Secure web service*, ACM, 2004, pp. 1–8.

[20] Cranor, L. F.; Reidenberg, J. R. Can user agents accurately represent privacy notices. In *30th Research Conference on Communication, Information and Internet Policy, Alexandria, VA*, 2002.

[21] Schunter, M.; Van Herreweghen, E.; Waidner, M. Expressive privacy promises-how to improve the platform for privacy preferences (p3p). In *Position paper for W3C Workshop on the Future of P3P (available at www. w3. org/2002/p3p-ws/pp/ibm-zuerich. pdf)*, 2002.

[22] Dean, M.; Schreiber, G.; Bechhofer, S.; et al. OWL web ontology language reference. *W3C Recommendation February*, volume 10, 2004.

[23] Group, W. O. W.; et al. {OWL} 2 Web Ontology Language Document Overview. 2009.

[24] Sirin, E.; Parsia, B. SPARQL-DL: SPARQL Query for OWL-DL. In *OWLED*, volume 258, 2007.

[25] Kremen, P.; Sirin, E. SPARQL-DL implementation experience. In *Proceedings of the 4th OWL Experiences and Directions DC Workshop (OWLED-DC-2008). CEUR Workshop Proceedings*, volume 496, 2008.

[26] Prud'hommeaux, E.; Seaborne, A. SPARQL Query Language for RDF. W3C recommendation, W3C, Jan. 2008, http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/.

[27] Knublauch, H.; Hendler, J.; Idehen, K. SPIN – Overview and Motivation. `http://www.w3.org/Submission/2011/SUBM-spin-overview-20110222/`, 2011, accessed: 2015-09-14.

[28] TopQuadrant. TobBraid Composer. `http://www.topquadrant.com/tools`, accessed: 2015-09-14.

[29] Jena, A. Apache jena. *jena. apache. org [Online]. Available: http://jena. apache. org [Accessed: Mar. 20, 2014]*, 2013.

[30] Stojanovic, L. *Methods and tools for ontology evolution*. Doctoral thesis, Karlsruhe Institute of Technology, 2004. Available from: `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000003270`

[31] Egana, M.; Antezana, E.; Stevens, R. Transforming the axiomisation of ontologies: The ontology pre-processor language. *Proceedings of OWLED*, 2008.

[32] Horridge, M.; Knublauch, H.; Rector, A.; et al. *A Practical Guide To Building OWL Ontologies With The Protege-OWL Plugin*. University of Manchester, first edition, 2004. Available from: `http://home.skku.edu/~samoh/class/sw/ProtegeOWLTutorial.pdf`

[33] Iannone, L.; Rector, A.; Stevens, R. Embedding knowledge patterns into owl. In *The Semantic Web: Research and Applications*, Springer, 2009, pp. 218–232.

[34] Corcho, Ó.; Roussey, C.; Blázquez, L. M. V.; et al. Pattern-based OWL Ontology Debugging Guidelines. In *Proceedings of the Workshop on Ontology Patterns (WOP 2009), collocated with the 8th International Semantic Web Conference (ISWC-2009), Washington D.C., USA, 25 October, 2009., CEUR Workshop Proceedings*, volume 516, edited by E. Blomqvist; K. Sandkuhl; F. Scharffe; V. Svátek, CEUR-WS.org, 2009. Available from: `http://ceur-ws.org/Vol-516/pap02.pdf`

[35] Chen, C.; Matthews, M. M. Extending description logic for reasoning about ontology evolution. In *Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence*, IEEE Computer Society, 2007, pp. 452–456.

[36] De Giacomo, G.; Lenzerini, M.; Rosati, R. On Higher-Order Description Logics. In *Description Logics*, 2009.

[37] Pan, J. Z.; Horrocks, I. OWL FA: a metamodeling extension of OWL D. In *Proceedings of the 15th international conference on World Wide Web*, ACM, 2006, pp. 1065–1066.

[38] Motik, B. On the properties of metamodeling in OWL. *Journal of Logic and Computation*, volume 17, no. 4, 2007: pp. 617–637.

[39] Gabbay, D. M.; Schmidt, R.; Szalas, A. *Second Order Quantifier Elimination: Foundations, Computational Aspects and Applications*. College Publications, 2008.

[40] Akroun, L.; Nourine, L.; Toumani, F. Reasoning in description logics with variables: preliminary results regarding the EL logic. In *Proceedings of the 28th International Workshop on Description Logics, Athens,Greece, June 7-10, 2015.*, *CEUR Workshop Proceedings*, volume 1350, edited by D. Calvanese; B. Konev, CEUR-WS.org, 2015. Available from: `http://ceur-ws.org/Vol-1350/paper-42.pdf`

[41] Colucci, S.; Di Noia, T.; Di Sciascio, E.; et al. Second-order description logics: Semantics, motivation, and a calculus. In *23rd International Workshop on Description Logics DL2010*, 2010, p. 67.

[42] Greiner, R.; Smith, B. A.; Wilkerson, R. W. A correction to the algorithm in Reiter's theory of diagnosis. *Artificial Intelligence*, volume 41, no. 1, 1989: pp. 79–88.

[43] Lin, L.; Jiang, Y. The computation of hitting sets: Review and new algorithms. *Information Processing Letters*, volume 86, no. 4, 2003: pp. 177–184.

[44] Pill, I.; Quaritsch, T. Optimizations for the Boolean Approach to Computing Minimal Hitting Sets. In *ECAI*, 2012, pp. 648–653.

[45] Abreu, R.; van Gemund, A. J. A Low-Cost Approximate Minimal Hitting Set Algorithm and its Application to Model-Based Diagnosis. In *SARA*, volume 9, 2009, pp. 2–9.

[46] Pill, I.; Quaritsch, T.; Wotawa, F. From conflicts to diagnoses: An empirical evaluation of minimal hitting set algorithms. In *22nd Int. Workshop on the Principles of Diagnosis*, 2011, pp. 203–210.

[47] van Bevern, R. Towards optimal and expressive kernelization for d-hitting set. *Algorithmica*, volume 70, no. 1, 2014: pp. 129–147.

[48] Suarez-Figueroa, M. C.; Gomez-Perez, A.; Fernandez-Lopez, M. The NeOn methodology for ontology engineering. In *Ontology engineering in a networked world*, Springer, 2012, pp. 9–34.

[49] Sure, Y.; Staab, S.; Studer, R. On-to-knowledge methodology (OTKM). In *Handbook on ontologies*, Springer, 2004, pp. 117–132.

[50] Uschold, M.; King, M. *Towards a methodology for building ontologies*. Citeseer, 1995.

[51] Blasko, M.; Kremen, P.; Kouba, Z. Ontology Evolution Using Ontology Templates. *Open Journal of Semantic Web (OJSW)*, volume 2, no. 1, 2015: pp. 15–28.

[52] Motik, B.; Patel-Schneider, P. F.; Parsia, B.; et al. OWL 2 web ontology language: Structural specification and functional-style syntax. *W3C recommendation*, volume 27, no. 65, 2009: p. 159.

[53] Sirin, E.; Parsia, B.; Grau, B. C.; et al. Pellet: A Practical OWL-DL Reasoner. *Web Semant.*, volume 5, no. 2, June 2007: pp. 51–53, ISSN 1570-8268, doi:10.1016/j.websem.2007.03.004. Available from: `http://dx.doi.org/10.1016/j.websem.2007.03.004`

[54] Křemen, P.; Mička, P.; Blaško, M.; et al. Ontology-driven mindmapping. In *Proceedings of the 8th International Conference on Semantic Systems*, ACM, 2012, pp. 125–132.

[55] Konev, B.; Lutz, C.; Walther, D.; et al. Semantic Modularity and Module Extraction in Description Logics. In *ECAI*, 2008, pp. 55–59.

[56] Harris, S.; Seaborne, A.; PrudâĂŹhommeaux, E. SPARQL 1.1 Query Language (2013). *W3C Recommendation*, 2014.

[57] Cacciotti, R.; Blaško, M.; Valach, J. A diagnostic ontological model for damages to historical constructions. *Journal of Cultural Heritage*, volume 16, no. 1, 2015: pp. 40–48.

[58] Marchiori, M. The Platform for Privacy Preferences 1.0 (P3P1.0) Specification. W3C recommendation, W3C, Apr. 2002, http://www.w3.org/TR/2002/REC-P3P-20020416/.

[59] Blaško, M.; Křemen, P.; Kouba, Z. Privacy Protection Using Semantic Technologies. In *Proceedings of the 12th European Meeting on Cybernetics and Systems Research*, Vienna, 2009.

[60] Schneider, P. *P3P editor based on semantic technologies*. Master's thesis, Czech Technical University in Prague, Prague, Czech Republic, 2012.

[61] Han, B.; Lee, S.-J. Deriving minimal conflict sets by CS-trees with mark set in diagnosis from first principles. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, volume 29, no. 2, 1999: pp. 281–286.

[62] Dürst, M.; Suignard, M. Internationalized resource identifiers (IRIs). Technical report, 2004, http://www.rfc-editor.org/rfc/rfc3987.txt.

[63] Horrocks, I.; Kutz, O.; Sattler, U. The Even More Irresistible SROIQ. *KR*, volume 6, 2006: pp. 57–67.

Brief Summary of the Thesis

Sophisticated information systems make often use of ontologies that express the semantics of data the system is processing. Such systems include various applications of semantic web and are using technologies of semantic web. However, it is a seldom case when an ontology stays in the same status for the whole life cycle of the application that makes use of it. Normally, the ontology needs to be modified – typically extended – during its life cycle. Such a process of permanent small changes made to the ontology is referred as an ontology evolution.

Evolving ontologies by domain experts is difficult and typically cannot be performed without the assistance of an ontology engineer. This process takes a long time and often recurrent modeling errors have to be resolved. This doctoral thesis proposes a technique for creating controlled ontology evolution scenarios that ensure the consistency of the possible ontology evolution and give guarantees to the domain expert that his/her updates do not cause inconsistency. We introduce ontology templates that formalize the notion of controlled evolution together with consistency checking algorithm. The ontology templates are reusable across multiple scenarios which helps ontology engineers to manage whole ontology evolution. In addition, the algorithm for the consistency check can be used for auxiliary reasoning service that provides to ontology engineers new options to check certain expectations of the evolving ontology. The specific contributions are: (1) framework and methodology for definition of ontology evolution scenarios, (2) algorithms to compute ontological implications of the evolution scenarios, (3) proposal of new non-standard reasoning service that implements the algorithms to support ontological engineering methods, (4) prototypical implementation of the service and its experimental evaluation, and (5) validation of the approach on real cases.

The research was motivated by the experience gained during two projects – Netcarity and MONDIS. In Netcarity, the domain was privacy protection. The P3P/Appel approach proposed by the WWW consortium has proven insufficient semantics. This is why a novel approach based on description logic was proposed within Netcarity project. It includes the transformation of P3P policy into an ontology, which can be viewed as an evolution of a generic privacy ontology. To ensure that the transformation process will result into a consistent ontology, the need of a controlled evolution process was discovered.

In the MONDIS project, the ontology developed in a close cooperation of a domain expert with a knowledge engineer needs to be enriched mainly with the knowledge of new protection techniques. However, form the manageability point of view, it is necessary to get rid off the presence of the knowledge engineer in the evolution process. The only applicable approach is to prepare evolution scenarios expressed in terms of ontology templates that allow the domain engineer himself to evolve the ontology in a consistent way.

The thesis demonstrate the practical meaning and applicability of the developed evolution framework on the two above introduce use cases. The thesis open also interesting topics for a future work. First, we would like to experiment with various ontology templates w.r.t. their usability for domain experts. Compared to ontology design patterns that are meant for ontology engineers (in the same sense as software design patterns are meant for software engineers), ontology templates are also meant for domain experts – thus, they have to be simpler, intuitive and have the formal guarantees introduced in this thesis. Second, it was shown that our framework can be used not only for providing consistency within the evolution scenario, but also for specific entailments of the ontology. In the P3P case, the framework was used to check satisfiability of all classes within the scenario. It would be interesting to provide an ontology engineer a service that would enhance these possibilities to check arbitrary entailments, even formulated in terms of ontology templates. Third, our evaluation of prototype service brought up new ideas for optimization of the provided algorithms.

Publications of the Author Relevant to the Thesis

Reviewed publication in journal with impact factor:

[1]   Cacciotti, R. - **Blaško, M. (30% contribution)** - Valach, J.: A Diagnostic Ontological Model for Damages to Historical Constructions. Journal of Cultural Heritage. 2015, vol. 16, no. 1, p. 40-48. ISSN 1296-2074. Available from http://dx.doi.org/10.1016/j.culher.2014.02.002.

The paper has been cited in:

- De Tommasi, Giambattista, Fabio Fatiguso, Mariella De Fino, and Albina Scioti. "Methodological guidelines, operation protocols and innovative techniques for assessment and control of the built heritage."

- Cursi, Stefano, Davide Simeone, and Ilaria Toldo. "A Semantic Web Approach for Built Heritage Representation. "Computer-Aided Architectural Design Futures. The Next City-New Technologies and the Future of the Built Environment. Springer Berlin Heidelberg, 2015. 383-401.

Reviewed publication in journal:

[2]   **Blaško, M. (80% contribution)** - Křemen, P. - Kouba, Z.: Ontology Evolution Using Ontology Templates. In Open Journal of Semantic Web (OJSW), RonPub, 2015. Accepted in September 2015, Available from https://www.ronpub.com/publications/OJSW_2015v2i1n03_Blasko.pdf.

Other publications:

[3]   **Blaško, M. (50% contribution)** - Křemen, P. - Kouba, Z.: Privacy Protection Using Semantic Technologies. In Proceedings of the Twentieth European Meeting on Cybernetics and Systems Research. Vienna: Austrian Society for Cybernetics Studies, 2010, p. 585-590. ISBN 978-3-85206-178-8.

[4]   **Blaško, M. (50% contribution)** - Křemen, P. - Kouba, Z.: Semantic Approach to Privacy Protection. In 3rd Annual European Semantic Technology Conference 2009 -Proceedings. Vienna: Semantic Technology Institute International, 2009.

[5]   **Blaško, M. (60% contribution)** - Cacciotti, R. - Křemen, P. - Kouba, Z.: Monument Damage Ontology. In Progress in Cultural Heritage Preservation. Heidelberg: Springer, 2012, p. 221-230. ISSN 0302-9743. ISBN 978-3-642-34233-2.

The paper has been cited in:

○ Nedvědová, Klára, and Robert Pergl. "Cultural Heritage and Floods."

○ Nedvědová, Klára, and Robert Pergl. "CULTURAL HERITAGE AND FLOODS RISK PREPAREDNESS."

[6] Křemen, P. - **Blaško (20% contribution)**, M. - Šmíd, M. - Kouba, Z. - Ledvinka, M. - et al.: MONDIS: Using Ontologies for Monument Damage Descriptions. In Znalosti 2014. Praha: VŠE, 2014, p. 66-69. ISBN 978-80-245-2054-4.

Remaining Publications of the Author

[1]   Křemen, P. - Kouba, Z. - **Blaško, M. (20% contribution)** : Semantic Annotation of Objects. In Handbook of Research on Social Dimensions of Semantic Technology and Web Services. Hershey, Pennsylvania: IGI Global, 2009, p .223-238. ISBN 978-1-60566-650-1.

The paper has been cited in:

- ○ Fouad, Khaled M., Mostafa A. Nofal, and Jehan A. Hasan. "Building Learning Repository based on Semantic and Shareable Learning Objects. "International Journal of Computer Applications 70.17 (2013): 34-42.

[2]   Křemen, P. - Mička, P. - **Blaško, M. (25% contribution)** - Šmíd, M.: Ontology-Driven Mindmapping. In Proceedings of the 8th International Conference on Semantic Systems. New York: ACM, 2012, p. 125-132. ISBN 978-1-4503-1112-0.

[3]   Wolf, A. - Mulholland, P. - Zdráhal, Z. - **Blaško, M. (25% contribution)** : Knowledge Modelling to Support Inquiry Learning Tasks. In Lecture Notes in Computer Science. Berlin: Springer, 2010, p. 198-209. ISSN 0302-9743. ISBN 978-3-642-15280-1.

The paper has been cited in:

- ○ Wang, Feng, et al. "Research on semantic-based knowledge service for cluster supply chain. "Computer Supported Cooperative Work in Design (CSCWD), 2012 IEEE 16th International Conference on. IEEE, 2012.

[4]   Cacciotti, R. - Valach, J. - Kuneš, P. - Čerňanský, M. - **Blaško, M. (10% contribution)** - et al.: Introduction to an Ontology-Driven Documentation System of Damages to Cultural Heritage. International Journal of Heritage in the Digital Era. 2014, vol. 3, no. 2, p. 255-270. ISSN 2047-4970.

[5]   Cacciotti, R. - Valach, J. - Kuneš, P. - Čerňanský, M. - Křemen, P. - **Blaško, M. (10% contribution)** - et al.: Monument Damage Information System (MONDIS): An Ontological Approach to Cultural Heritage Documentation. In ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences. Goetingen: Copernicus GmbH, 2013, p. 55-60. ISSN 2194-9042.

The paper has been cited in:

○ Noardo, Francesca, and Antonia Spanó. "Towards a Spatial Semantic Management for the Intangible Cultural Heritage. "International Journal of Heritage in the Digital Era 4.2 (2015): 133-148.s."

○ Andaroodi, Elham, and Asanobu Kitamoto. "Online Ontology-Based Knowledge Representation of Historic Buildings: Publishing an RDF-Based Website. "International Journal of Heritage in the Digital Era 3.3 (2014): 475-500.

Participation of the Author in Projects and Software

[1]   Bierfoff, I. - Goosen, B. - Wintjens, K. - Huijnen, C. - Křemen, P. - et al.: Participatory Design of Netcarity Services Using Different Perspectives. In Proceedings of the first International AEGIS Conference. Seville: AEGIS project, 2010, p. 202-209.

[2]   Kouba, Z. - Valach, J. - Křemen, P. - Cacciotti, R. - Blaško, M. - Šmíd, M. - Kostov, B. - et al.: Defects in immovable cultural heritage objects: a knowledge-based system for analysis, intervention planning and prevention, NO. DF11P01OVV002 of the Ministry of Culture of the Czech Republic.

[3]   Blaško, M. - Šmíd, M. - Mička, P. - Kostov, B. - Křemen, P. - Kouba, Z.: "Efektivní tvorba znalostních systémů s využitím pokročilých sémantických technologií", Studentská grantová soutež ČVUT, 2010, (SGS10/276/OHK3/3T/13).

[4]   Křemen, P. - Blaško, M.: Terminology Editor. [RIV conditions fulfilling software (sooner authorized)]. 2014.

A Represenation of Ontology Temlate in SPARQL-based Controlled Evolution

Example of ontology template

Abstract $\mathcal{ALC}_x$ syntax

$T_1 = \{\$COMPONENT \sqsubseteq \forall hasMaterial.\neg\$MATERIAL\ \}$

Related SPARQL construct query

```
CONSTRUCT {
  $component rdfs:subClassOf
                  [ a owl:Restriction ;
                    owl:onProperty cms:hasMaterial ;
                    owl:allValuesFrom
                      [ a owl:Class ;
                        owl:complementOf $material ;
                      ];
                  ] .
  $component a owl:Class .
  $material a owl:Class .
  cms:hasMaterial a owl:ObjectProperty .
}
WHERE {
    $component (rdfs:subClassOf)+ cms:Component .
    $material (rdfs:subClassOf)+ cms:Material .
}
```

Representation of the template

```
:ot-component-not-possible-material
  rdfs:label "Not possible material of component"@en ;
  rdf:type owl:NamedIndividual ;
  ot-core:hasSpinQueryForConterpart [
```

```
rdf:type sp:Construct ;
sp:templates (
    [
      sp:object _:b78525 ;
      sp:predicate rdfs:subClassOf ;
      sp:subject sp:_component ;
    ]
    [
      sp:object owl:Restriction ;
      sp:predicate rdf:type ;
      sp:subject _:b78525 ;
    ]
    [
      sp:object cms:hasMaterial ;
      sp:predicate owl:onProperty ;
      sp:subject _:b78525 ;
    ]
    [
      sp:object _:b31822 ;
      sp:predicate owl:allValuesFrom ;
      sp:subject _:b78525 ;
    ]
    [
      sp:object owl:Class ;
      sp:predicate rdf:type ;
      sp:subject _:b31822 ;
    ]
    [
      sp:object sp:_material ;
      sp:predicate owl:complementOf ;
      sp:subject _:b31822 ;
    ]
    [
      sp:object owl:Class ;
      sp:predicate rdf:type ;
      sp:subject sp:_component ;
    ]
    [
      sp:object owl:Class ;
      sp:predicate rdf:type ;
      sp:subject sp:_material ;
    ]
    [
```

```
                sp:object owl:ObjectProperty ;
                sp:predicate rdf:type ;
                sp:subject cms:hasMaterial ;
              ]
           ) ;
        sp:text """CONSTRUCT {
  $component rdfs:subClassOf
                   [ a owl:Restriction ;
                     owl:onProperty cms:hasMaterial ;
                     owl:allValuesFrom
                       [ a owl:Class ;
                         owl:complementOf $material ;
                       ];
                   ] .
  $component a owl:Class .
  $material a owl:Class .
  cms:hasMaterial a owl:ObjectProperty .
}
WHERE {
    $component (rdfs:subClassOf)+ cms:Component .
    $material (rdfs:subClassOf)+ cms:Material .
}"""^^xsd:string ;
        sp:where (
            [
              rdf:type sp:TriplePath ;
              sp:object cms:Component ;
              sp:path [
                  rdf:type sp:ModPath ;
                  sp:modMax -2 ;
                  sp:modMin 1 ;
                  sp:subPath rdfs:subClassOf ;
                ] ;
              sp:subject sp:_component ;
            ]
            [
              rdf:type sp:TriplePath ;
              sp:object cms:Material ;
              sp:path [
                  rdf:type sp:ModPath ;
                  sp:modMax -2 ;
                  sp:modMin 1 ;
                  sp:subPath rdfs:subClassOf ;
                ] ;
```

```
            sp:subject sp:_material ;
          ]
        ) ;
    ] ;
  rdfs:subClassOf :MondisOTI ;
.
```